A Symbolic Algorithm for Shortest EG Witness Generation

Yang Zhao and Xiaoqing Jin and Gianfranco Ciardo
Department of Computer Science and Engineering,
University of California, Riverside
Email: {zhaoy, jinx, ciardo}@cs.ucr.edu

***Abstract*—Witness generation is a fundamental model checker feature, but generating shortest witnesses for an EG CTL formula has long been a difficult problem of both theoretical and practical relevance. We propose a symbolic approach to shortest EG witness generation based on edge-valued multi-way decision diagrams. We employ a fixpoint symbolic iteration to compute the transitive closure enhanced with distance information, using the saturation algorithm to cope with the high computational complexity of this approach. We also extend this approach to tackling the shortest witness generation for other properties and the shortest fair witness generation. Experimental results show that our approach can generate a shortest witness which could not be found within acceptable time using previous algorithms.**

I. Introduction

Model checking aims at rigorously verifying the conformance of a system to a given temporal property. Instead of merely giving "true" or "false" answers, current model checkers also return a possible execution of the system under verification, called a *witness* or *counterexample*, which, respectively, *validates* or *violates* the property. Since the violation of an expected property often reflects design errors in the system, counterexample generation is a valuable feature of model checkers which helps debugging. Theoretically, it is interesting to investigate practical algorithms that can generate a shortest witness or counterexample. Practically, a shorter counterexample is naturally more readable and helpful for engineers to understand and correct erroneous behavior in the system. Thus, both theory and engineering requirements motivate research in shortest witness generation.

Computational tree logic (CTL) [1] is widely used because of its simple yet expressive semantics. As we review in Section II-A, counterexample generation to a universal property can be converted to an equivalent witness generation to an existential property, in the context of CTL. Thus, we only focus on shortest witness generation. Many efforts [2][3][4][5] have been made on this topic. Unlike EX or EU witnesses, which are finite paths, EG witnesses are "lasso-shaped" [2], i.e., they contain a prefix leading to a cycle. Locating this cycle is the crucial step in witness generation, and it is even more difficult to find a shortest (length of handle plus cycle) witness.

The approach in this paper falls into the class of symbolic algorithms, since it employs decision diagrams. Specifically, we utilize Edge-Valued Multi-way Decision Diagram (EVMDD) to encode the Transitive Closure with Distance (TCD), and employ an advanced algorithm for symbolic exploration, saturation [6], to efficiently build the EVMDD encoding the TCD.

The remainder of this paper is structured as follow: Section II introduces the relevant background on CTL and the symbolic data structure we use. Section III presents our approach, stressing the computation of the TCD using the saturation algorithm. Section IV extends our approach to a more general class of shortest witness generation problems. Section V offers some experimental results. We conclude this paper and outline future work in the last section.

II. Preliminaries

A Kripke structure with disjunctively-partitioned next-state function is a tuple $(\mathcal{S}, \mathbf{s}_{init}, \mathcal{E}, \{\mathcal{N}_\alpha : \alpha \in \mathcal{E}\}, \mathcal{A}, \mathcal{L})$ where:

- $\mathcal{S}$ is the potential state space given by the cross-product $\mathcal{S}_L \times \cdots \times \mathcal{S}_1$ of the local state spaces of L (finite) submodels. Thus, each (global) state $\mathbf{i}$ is a tuple $(i_L, ..., i_1)$, where $i_k \in \mathcal{S}_k$ is a local state of the k^{th} submodel.
- $\mathbf{s}_{init} \in \mathcal{S}$ is the (single) initial state.
- $\mathcal{E}$ is a set of (asynchronous) events.
- $\mathcal{N}_\alpha : \mathcal{S} \to 2^{\mathcal{S}}$ is the next-state function for event $\alpha \in \mathcal{E}$, so that $\mathcal{N}_\alpha(\mathbf{i})$ is the set of states that can be nondeterministically reached in one step when event α fires in state $\mathbf{i}$. We let $\mathcal{N}_\alpha(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}_\alpha(\mathbf{i})$ and $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$.
- $\mathcal{A}$ is a set of atomic propositions.
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{A}}$ is a labeling function listing the atomic propositions that hold in each state. Let $\mathcal{S}_p$ be the set of states where p holds, $\{\mathbf{s} \in \mathcal{S} : p \in \mathcal{L}(\mathbf{s})\}$.

Throughout this paper, $\mathbf{i}, \mathbf{j}$, and $\mathbf{k}$ denote single states; $\mathcal{X}$ and $\mathcal{Y}$ denote sets of states; α and β denote events; and f, p, and q denote atomic propositions. Also, while we assume a single initial state, our algorithm can be applied to systems with multiple initial states by simply adding a new "virtual initial state" $\mathbf{s}_{init}$ and a new event with auxiliary transitions from $\mathbf{s}_{init}$ to each initial state of the original system. Then, for EGp, we let $\mathbf{s}_{init} \in \mathcal{S}_p$ and the shortest witness generated for the modified system is equivalent to that of the original system except for the first auxiliary transition.

The basic idea of symbolic algorithms is to map a set of states to a symbolic structure; we use a multi-way decision diagram (MDD) [7]. The state-space exploration can be executed implicitly by MDD manipulation instead of explicitly traversing the transition graph. For simplicity, we do not

differentiate the notation between a set of states $\mathcal{X}$ or next-state function $\mathcal{N}_\alpha$ and their MDD encoding; what is meant is easily determined by the context.

A. Background and related work

We assume readers are familiar with the semantics of CTL [1]. Only a generator subset of CTL operators, e.g., $\{\mathsf{EX}, \mathsf{EU}, \mathsf{EG}\}$, needs to be implemented in a model checker, as the all other CTL operators can be expressed using the generator set and ordinary boolean operators. The correspondence between counterexamples to universal "A" properties and witnesses to existential "E" properties is as follows:

a counterexample to	is the same as a witness to
$\mathsf{AX}p$	$\mathsf{EX}(\neg p)$
$\mathsf{AG}p$	$\mathsf{EF}(\neg p)$
$\mathsf{A}[p\mathsf{U}q]$	$\mathsf{E}[\neg q\mathsf{U}(\neg p \wedge \neg q)] \vee \mathsf{EG}(\neg q)$
$\mathsf{AF}p$	$\mathsf{EG}(\neg p)$

Clarke et al. [2] proposed the first symbolic approach to CTL witness generation. Using symbolic breath-first search, witness generations for EX and EU can naturally guarantee minimality, but the problem is much more difficult for the EG operator. A witness to $\mathsf{EG}p$ is composed of a path from the initial state to a cycle, such that all states along that path and on the cycle satisfy p. According to CTL semantics, if a state satisfies $\mathsf{EG}p$, it must have a successor that also satisfies $\mathsf{EG}p$. Thus, we can incrementally build a path of states satisfying $\mathsf{EG}p$, which must finally lead to a state already on the path, closing the cycle and resulting in a witness. Figure 1 shows the pseudocode of this algorithm, which can be easily implemented using BDDs or MDDs. A witness generation algorithm for weakly fair EG was also proposed in [2] based on this idea. Since there might be multiple successors satisfying $\mathsf{EG}p$, the algorithm is nondeterministic and the length of the witness depends in general on which state is chosen at each step. We stress that, while the pseudocode uses a symbolic encoding, this algorithm is largely explicit, as it follows a single specific path. Decision diagrams help very little, especially if $\mathcal{N}(\mathbf{i})$ is a very small set for each $\mathbf{i}$.

The work most related to ours was presented by Schuppan et al. in [3][8], which proposed a framework to convert a liveness property to a reachability problem by performing a *state-recording translation*, so that a shortest witness for $\mathsf{EG}p$ can then be generated using breadth-first search (BFS). The bottleneck of this approach mainly lies in the BFS over the quadratic state space of the original system. Our previous work [4] has shown that saturation can effectively speed up shortest trace generation. This paper extends the idea of [4] to the generation of shortest $\mathsf{EG}p$ witness. Instead of exploring a quadratic state space using BFS like [8], we employ saturation, usually more efficient than BFS for asynchronous systems, to build the TCD encoded as an EVMDD.

Orthogonal to symbolic algorithms, explicit-state model checkers adopt techniques such as heuristic-guided search [9] and crucial event identification [5] to shorten the generated witnesses. These techniques aim at achieving both an efficient of state-space exploration and shorter witnesses. Still, none of these algorithms is guaranteed to find a shortest EG witness.

```
EGwitnessBFS(stateset P) is    • P is the MDD encoding S_p
 1 stateset Q ← EG(P);         • witness exists only if s_init ∈ Q
 2 stateset X ← ∅;             • set of states in the witness
 3 stateset Y ← {s_init};      • current frontier
 4 while (X ∩ Y = ∅) do
 5   state i ← pick(Y);   print i;
 6   X ← X ∪ {i};   Y ← N(i) ∩ Q;
 7 endwhile
 8 i ← pick(X ∩ Y);   print i;
```

Fig. 1. BFS-based algorithm to generate a witness for $\mathsf{EG}p$.

B. Edge-valued multi-way decision diagrams

While BDDs and MDDs provide a compact encoding for Boolean functions, we often need to represent integer functions defined on a large set of states, such as the distance function which maps each state to its distance from the initial states. Widely used Multi-terminal Binary Decision Diagrams (MTBDDs, and the similar Algebraic Decision Diagrams, ADDs) [10] or Multi-terminal Multi-way Decision Diagrams (MTMDDs) are natural extensions of BDDs and MDDs to integer ranges. In this paper, however, we employ (additive) Edge-Valued Multi-way Decision Diagrams (EVMDDs) [4], which have the advantage of being more compact: [11] proves that an EVMDD never contains more nodes than the equivalent MTMDD under the same variable order. Furthermore, the saturation algorithm can be used on EVMDDs to speed up the fixpoint iterations, as [4] showed for the computation of distance functions, and as we show in Section III for the computation of TCD.

Among the many variants of EVMDDs, we choose a specific additive version, EV$^+$MDD, whose normalization rules allow us to encode partial integer functions where "undefined" has the semantics of "∞". Given L sets $\mathcal{S}_k = \{0, ..., n_k - 1\}$, for $L \geq k \geq 1$, an EV$^+$MDD over $\mathcal{S} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1$ is a directed acyclic level-oriented edge-labeled multi-graph where:

- Ω is the only *terminal* node and is at level 0. We write $\Omega.lvl = 0$.
- A nonterminal node a is at some level k, with $L \geq k \geq 1$, and has n_k outgoing edges, each labeled with a different index $i_k \in \mathcal{S}_k$, pointing to a lower-level node b, and associated with a value $\rho \in \mathbb{N} \cup \{\infty\}$. We write $a.lvl = k$, $a[i_k] = \langle \rho, b \rangle$, $b = a[i_k].node$, and $\rho = a[i_k].val$.
- There is a single root node r^*, at level L, with a "dangling" incoming edge associated with a value $\rho^* \in \mathbb{N}$. We write $\langle \rho^*, r^* \rangle$ to denote the entire EV$^+$MDD.

For a *canonical* form of EV$^+$MDDs, assumed from now on, we require that [4]:

- If $a[i] = \langle \infty, b \rangle$, then $b = \Omega$.
- If $a[i] = \langle \rho, b \rangle$ with $\rho \in \mathbb{N}$, then $b.lvl = a.lvl - 1$.
- For each node a at level $k > 0$, there is an index $i_k \in \mathcal{S}_k$ s.t. $a[i_k].val = 0$.
- There are no *duplicate* nodes, i.e., given two distinct nodes a and b at level $k > 0$, there is an index $i_k \in \mathcal{S}_k$

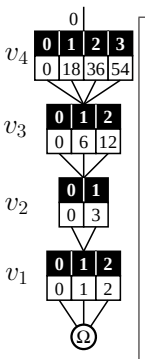

Fig. 2. An EVMDD.

```
evmdd Normalize(evmdd.node a)
 1 if a = Ω then return ⟨0,Ω⟩;
 2 level k ← a.lvl;
 3 b ← NewNode(k); • create a node at level k
 4 for i_k ∈ {0, ..., n_k − 1} do
 5   ⟨μ,s⟩ ← Normalize(a[i_k].node);
 6   b[i_k] ← ⟨μ + a[i_k].val,s⟩;
 7 endfor
 8 ρ ← min{b[i_k].val : i_k ∈ {0, ..., n_k − 1}}
 9 for i_k ∈ {0, ..., n_k − 1} do
10   b[i_k].val ← b[i_k].val − ρ
11 endfor
12 return ⟨ρ,b⟩;
```

Fig. 3. Pseudocode for $Normalize$.

s.t. $a[i_k] \neq b[i_k]$.

An edge $\langle \rho,a \rangle$ encodes the function $f_{\langle \rho,a \rangle} : \mathcal{S}_k \times \cdots \times \mathcal{S}_1 \to \mathbb{N} \cup \{\infty\}$, where $a.lvl = k$, defined recursively as follows:

$$f_{\langle \rho,a \rangle}(i_k, ..., i_1) = \begin{cases} \rho & \text{if } a = \Omega \\ \rho + f_{a[i_k]}(i_{k-1}, ..., i_1) & \text{otherwise.} \end{cases}$$

For example, Figure 2 shows a canonical EV$^+$MDD encoding $f(v_4, v_3, v_2, v_1) = 18v_4 + 6v_3 + 3v_2 + v_1$, where $v_4 \in \{0, 1, 2, 3\}; v_3, v_1 \in \{0, 1, 2\}; v_2 \in \{0, 1\}$. The above definition allows us to canonically encode any function from $\mathcal{S}$ to $\mathbb{N} \cup \{\infty\}$ with the exception of the constant function identically equal to ∞, for which we allow the special encoding $\langle \infty,\Omega \rangle$.

In a practical implementation, duplicated nodes are avoided by maintaining a *unique table UT*, and each node is *normalized* before inserting it in the unique table to conform to the canonical rule. Function $Normalize$ in Figure 3 describes the procedure to normalize an EV$^+$MDD; most manipulation functions, however, work with and maintain normalized EV$^+$MDDs, so only newly created nodes require normalization prior to inserting them in the unique table.

Operations on integer functions can be efficiently implemented by EV$^+$MDD. Three EV$^+$MDD operations are extensively used in our work:

- Min: given two EV$^+$MDDs $\langle \rho,a \rangle$ and $\langle \sigma,b \rangle$, return an EV$^+$MDD $\langle \mu,r \rangle$ encoding function $\min(f_{\langle \rho,a \rangle}, f_{\langle \sigma,b \rangle})$, where $\min(\infty, \rho) = \rho$ is a special case.
- Sum: given two EV$^+$MDDs $\langle \rho,a \rangle$ and $\langle \sigma,b \rangle$, return an EV$^+$MDD $\langle \mu,r \rangle$ encoding function $f_{\langle \rho,a \rangle} + f_{\langle \sigma,b \rangle}$, where $\infty + \rho = \infty$ is a special case.
- $MinState$: given an EV$^+$MDD $\langle \rho,a \rangle$, return a state $\mathbf{i}$ such that $f_{\langle \rho,a \rangle}(\mathbf{i})$ is the minimum value of function $f_{\langle \rho,a \rangle}$.

Figure 4 shows procedures Min and Sum. These algorithms run recursively on each level, without having to enumerate every state of a potentially huge state space. It is known [10] that the number of recursive calls of the generic *Apply* operation, including Min and Sum, for EVMDDs at most equals those for MTMDDs representing the same function. Procedure $MinState$ is even simpler: since each nonterminal node must have at least one edge with an associated value of 0, we can simply follow any 0-value path from the root $\langle \rho^*,r^* \rangle$ to Ω. The function encoded by the EV$^+$MDD evaluates to the minimum possible value, ρ^*, for any state corresponding to such a path.

```
evmdd Min(evmdd ⟨ρ,p⟩, evmdd ⟨σ,q⟩)
 1 int μ ← min{ρ, σ};   level k ← p.lvl;
 2 if ρ = ∞ then return ⟨σ,q⟩;
 3 if σ = ∞ then return ⟨ρ,p⟩;
 4 if p = q then return ⟨μ,p⟩; • includes the case k = 0, i.e. p = q = Ω
 5 if InCache_Min(⟨p, q, ρ−σ, r⟩) then return ⟨μ,r⟩; • Assume ρ ≥ σ, if not, swap two parameters.
 6 node r ← NewNode(k);
 7 for i_k ∈ S_k do
 8   r[i_k] ← Min(⟨ρ − μ + p[i_k].val,p[i_k].node⟩,
     ⟨σ − μ + q[i_k].val,q[i_k].node⟩);
 9 endfor
10 ⟨γ,r⟩ ← Normalize(r);
11 InsertUT(r);   CacheAdd_Min(⟨p, q, ρ − σ, r⟩);
12 return ⟨μ + γ,r⟩;
```

```
evmdd Sum(evmdd ⟨ρ,p⟩, evmdd ⟨σ,q⟩)
 1 int μ ← ρ + σ;   level k ← p.lvl;
 2 if ρ = ∞ or σ = ∞ then return ⟨∞,Ω⟩;
 3 if InCache_Sum(⟨p, q, ⟨γ,r⟩⟩) then return ⟨μ + γ,r⟩;
 4 node r ← NewNode(k);
 5 for i_k ∈ S_k do
 6   r[i_k] ← Sum(p[i_k], q[i_k]);
 7 endfor
 8 ⟨γ,r⟩ ← Normalize(r);
 9 InsertUT(r);   CacheAdd_Sum(⟨p, q, ⟨γ,r⟩⟩);
10 return ⟨μ + γ,r⟩;
```

Fig. 4. Pseudocode for Min and Sum.

III. Shortest EG witness generation

Section III-A provides an overview of our algorithm for shortest EG witness generation. Then, Section III-B describes the computation of TCD, the most critical step.

A. Overview

A witness to EGp in a finite-state system, often described as *lasso-shaped witness* [12], is an infinite path consisting of a finite prefix leading to a cycle, We provide the following definition to discuss EG witnesses in more detail:

Definition 1: Given $p \in \mathcal{A}$, a finite acyclic path $\pi_s = \mathbf{s}_{init} \to \mathbf{i}_1 \to \cdots \to \mathbf{i}_n$ is a p-stem of length $n \geq 0$ if $\mathbf{s}_{init}, \mathbf{i}_1, \ldots, \mathbf{i}_n \in \mathcal{S}_p$; a (cyclic) path $\pi_c : \mathbf{i}_1 \to \cdots \to \mathbf{i}_m \to \mathbf{i}_1$ is a p-cycle of length $m \geq 1$ if $\mathbf{i}_1, \ldots, \mathbf{i}_m \in \mathcal{S}_p$. Let $stem(p, \mathbf{i})$ be the set of p-stems that terminate in state $\mathbf{i}$ and $cycle(p, \mathbf{i})$ be the set of p-cycles that start in state $\mathbf{i}$; when p is understood, we simply write $stem(\mathbf{i})$ and $cycle(\mathbf{i})$. □

An EGp witness is composed of a p-stem leading to a p-cycle. We say that state $\mathbf{k}$ which terminates a p-stem and also starts a p-cycle is a *knot*. The initial state is the knot in witnesses which have the 0-length stems. If EGp holds in the initial state, a state $\mathbf{k} \in \mathcal{S}_p$ induces a set of witnesses $witness(\mathbf{k}) = stem(\mathbf{k}) \times cycle(\mathbf{k})$, i.e., all combinations of p-stems in $stem(\mathbf{k})$ and p-cycles in $cycle(\mathbf{k})$. A shortest witness among $witness(\mathbf{k})$ consists of a shortest $stem(\mathbf{k})$ and a shortest $cycle(\mathbf{k})$. Hence, finding the shortest EGp witness

can be seen as the minimization problem (let $|\pi|$ be the length of path π):

$$\min_{\mathbf{k}\in\mathcal{S}_p}\left(\min_{\pi_s\in stem(\mathbf{k})}(|\pi_s|)+\min_{\pi_c\in cycle(\mathbf{k})}(|\pi_c|)\right) \quad (1)$$

Given a knot state $\mathbf{k}$, algorithms in [2], [4] can efficiently find the shortest $stem(\mathbf{k})$ and $cycle(\mathbf{k})$. The difficulty lies in finding a knot state $\mathbf{k}^*$ that induces a globally shortest witness in Equation 1. This difficulty can be attributed to the lack of algorithms able to compute the distance information between pairs of states in a huge state space. We attack the problem from this angle by computing TCD.

Definition 2: (Transitive Closure with Distance). Function $TCD_p : \mathcal{S}\times\mathcal{S}\to\mathbb{N}^+\cup\{\infty\}$ is such that $TCD_p(\mathbf{i},\mathbf{j})$ is the length of a non-zero shortest path $\mathbf{i}\to\mathbf{s}_1\to\mathbf{s}_2\to\cdots\to\mathbf{j}$ where $\mathbf{i},\mathbf{s}_1,\mathbf{s}_2,\ldots,\mathbf{j}\in\mathcal{S}_p$, and $TCD_p(\mathbf{i},\mathbf{j})=\infty$ if no such path exists. Let $TCD_p^{triv}(\mathbf{i},\mathbf{j})$ be an extension of $TCD_p(\mathbf{i},\mathbf{j})$ that:

$$TCD_p^{triv}(\mathbf{i},\mathbf{j})=\begin{cases}0 & \text{if } \mathbf{i}=\mathbf{j}\\ TCD_p(\mathbf{i},\mathbf{j}) & \text{otherwise}\end{cases}$$

Also, let $TCD_p^{-1}(\mathbf{i},\mathbf{j})=TCD_p(\mathbf{j},\mathbf{i})$. □

As the base case, $TCD_p(\mathbf{i},\mathbf{j})=1$ if $\mathbf{i}$ and $\mathbf{j}$ satisfy p and $\mathbf{j}\in\mathcal{N}(\mathbf{i})$. Define

$$TCD_p^{stem}(\mathbf{i}) \triangleq TCD_p^{triv}(\mathbf{s}_{init},\mathbf{i})$$
$$TCD_p^{cycle}(\mathbf{i}) \triangleq TCD_p(\mathbf{i},\mathbf{i})$$

Formula 1 can then be rewritten as:

$$\min_{\mathbf{k}\in\mathcal{S}_p}(TCD_p^{stem}(\mathbf{k})+TCD_p^{cycle}(\mathbf{k})),$$

TCD_p is a two-parameter function, thus must be encoded with a $2L$-level EV$^+$MDD, while TCD_p^{stem} and TCD_p^{cycle} are single-parameter functions which can be encoded with L-level EV$^+$MDDs and are obtained from TCD_p through symbolic manipulation. Thus, our algorithm for EG witness generation consists of the following steps:

1) Build the $2L$-level EV$^+$MDD encoding TCD_p.
2) Build L-level EV$^+$MDDs encoding TCD_p^{stem} and TCD_p^{cycle}. Compute the sum of these two EV$^+$MDDs, which encodes length of the shortest witness induced by each state $\mathbf{k}\in\mathcal{S}_p$.
3) Extract a knot state that achieves the minimum value in the resulting function.
4) Find the shortest paths from the initial state to the knot (p-stem) and from the knot to itself (p-cycle). These two paths form a shortest witness for EGp.

As the computation of TCD_p is the most time and memory intensive step in the overall procedure, we discuss it in detail in next section.

B. Computing TCD

Building TCD_p is essentially the classic all-pair shortest path problem in a modified graph from the original discrete-state system where only states (vertices) in $\mathcal{S}_p$ and transitions (edges) between these states are retained. All edges have unit weight, so the distance between $\mathbf{i}$ and $\mathbf{j}$ equals $TCD_p(\mathbf{i},\mathbf{j})$. Instead of using a distance matrix, however, we utilize a $2L$-level EV$^+$MDD $\langle\tau,t\rangle$ to encode the distance between each pair of states:

$$f_{\langle\tau,t\rangle}(i_L,j_L,i_{L-1},j_{L-1},\ldots,i_1,j_1)=TCD_p(\mathbf{i},\mathbf{j}),$$

where $\mathbf{i},\mathbf{j}\in\mathcal{S}_p$, and we interleave the levels for $\mathbf{i}$ and $\mathbf{j}$. To correlate local states with their submodels, the levels of i_k and j_k in $2L$-level EV$^+$MDD are referred to by k_{2l} and k'_{2l}, respectively (unprimed and primed levels are interleaved in our implementation), and we let $Unprimed(k_{2l})=Unprimed(k'_{2l})=k$.

The procedure to build $\langle\tau,t\rangle$ is analogous to a symbolic implementation of Dijkstra's algorithm. We start from the EV$^+$MDD $\langle\tau_1,t_1\rangle$ encoding

$$f_{\langle\tau_1,t_1\rangle}(\mathbf{i},\mathbf{j})=\begin{cases}1 & \text{if } \exists\alpha\in\mathcal{E},\ \mathbf{j}\in\mathcal{N}_\alpha(\mathbf{i})\\ \infty & \text{otherwise}\end{cases}$$

(so that $\tau_1=1$ and all values associated with outgoing edges are either 0 or ∞), and use $\mathcal{N}_\alpha$ to build a new EV$^+$MDD $\mathcal{N}_\alpha(\langle\tau,t\rangle)$ satisfying

$$f_{\mathcal{N}_\alpha(\langle\tau,t\rangle)}(\mathbf{i},\mathbf{j})=\min\left(\min_{\mathbf{k}\in pre(\mathbf{j})}\left(f_{\langle\tau,t\rangle}(\mathbf{i},\mathbf{k})\right),\infty\right)$$

where $pre(\mathbf{j})=\{\mathbf{k}\in\mathcal{S}_p:\mathbf{j}\in\mathcal{N}_\alpha(\mathbf{k})\}$. Then, $\langle\tau,t\rangle$ can be updated:

$$\langle\tau,t\rangle\leftarrow Min(\langle\tau,t\rangle,\mathcal{N}_\alpha(\langle\tau,t\rangle)+1)$$

We can iteratively update $\langle\tau,t\rangle$ for any event $\alpha\in\mathcal{E}$, until achieving convergence. It is easy to prove that this procedure always terminates and that the fixpoint is the answer to the all-pair shortest path problem, regardless of the order of updates, as long as all next-state functions are considered often enough. However, different orders might lead to huge variations in the size of the EV$^+$MDDs encoding the intermediate results, as well as the runtime. Saturation [6] has been shown to be an effective fixpoint iteration scheme that tends to minimize peak memory consumption and accelerate convergence. We first partition the events $\mathcal{E}$ into $\{\mathcal{E}_L,\ldots,\mathcal{E}_1\}$, where $\alpha\in\mathcal{E}_k$ iff k is the *lowest* level satisfying:

- if α is enabled (disabled) in $(i_L,\ldots,i_1)\in\mathcal{S}$, then it is enabled (disabled) in any $(j_L,\ldots,j_{k+1},i_k,\ldots,i_1)\in\mathcal{S}$,
- α does not change any local state at levels above k, so that $\mathcal{N}_\alpha(i_L,\ldots,i_1)$ can be rewritten as $\{(i_L,\ldots,i_{k+1})\}\times\mathcal{N}_\alpha(i_k,\ldots,i_1)$.

Let $\mathcal{N}_k=\bigcup_{\alpha\in\mathcal{E}_k}\mathcal{N}_\alpha$, then $\mathcal{N}_k(\mathcal{X})$ can be computed only on nodes at level or below k, evidencing the benefit of *locality*, which is widely enjoyed by asynchronous systems. Analogously, $\mathcal{N}_k(\langle\tau,t\rangle)$ can also be computed locally and $\langle\tau,t\rangle$ can be updated solely considering nodes at or below level k_{2l}, instead of recomputing the overall EV$^+$MDD. Moreover, we can repeatedly update $\langle\tau,t\rangle$ using $\mathcal{N}_k,\ldots,\mathcal{N}_1$ until convergence, at which point we say that the nodes are *saturated* on level k. $\langle\tau,t\rangle$ is *saturated* on level k iff

$$\forall j\le k,\langle\tau,t\rangle\equiv Min(\langle\tau,t\rangle,\mathcal{N}_j(\langle\tau,t\rangle)+1)$$

```
ComputeTCD(mdd a)                                   • a encodes S_p
 1  evmdd ⟨1,t_0⟩ ← EVMDDencode(N);
 2  return TCDSat(a, ⟨1,t_0⟩);

evmdd TCDSat(mdd a, evmdd ⟨μ,n⟩)
 1  if n = Ω then return ⟨μ,Ω⟩;
 2  if InCache_TCDSat(a, n, ⟨λ,r⟩) then return ⟨λ + μ,r⟩;
 3  level k ← n.lvl;
 4  node t ← NewNode(k);
 5  mdd r ← N_Unprimed(k)
 6  for i, j ∈ S_Unprimed(k) s.t. n[i][j].val ≠ ∞ do
 7    if a[j] ≠ 0 then                              • constrain the path in a
 8      t[i][j] ← TCDSat(a[j], n[i][j]);
 9    else
10      t[i][j] ← n[i][j];
11    endif
12  endfor
13  for i ∈ S_Unprimed(l) s.t. n[i].val ≠ ∞ do
14    repeat
15      for j,j' ∈ S_l s.t. n[i][j].val ≠ ∞ ∧ r[j][j'].node ≠ 0 do
16        if a[j'] ≠ 0 then                         • constrain the path in a
17          ⟨η,u⟩ ← TCDRelProdSat(a[j'], n[i][j], r[j][j']);
18          t[i][j'] ← Min(t[i][j'], ⟨η + 1,u⟩);   • incr. distance
19        endif
20      endfor
21    until ⟨λ,t⟩ does not change;
22  endfor
23  ⟨λ,t⟩ ← Normalize(t);
24  InsertUT(t);    CacheAdd_TCDSat(a, n, ⟨λ,t⟩);
25  return ⟨λ + μ,t⟩;

evmdd TCDRelProdSat(mdd a, evmdd ⟨μ,n⟩, mdd r)
 1  if n = Ω then return ⟨μ,Ω⟩;                    • r = 1 in this case
 2  if InCache_TCDRelProdSat(a, n, r, ⟨λ,t⟩) then
 3    return ⟨λ + μ,t⟩;
 4  level k ← n.lvl;    node t ← NewNode(k);
 5  for i ∈ S_Unprimed(k) s.t. n[i].val ≠ ∞ do
 6    for j, j' ∈ S_Unprimed(k) s.t. n[i][j].val ≠ ∞ ∧ r[j][j'] ≠ 0 do
 7      if a[j'] ≠ 0 then                           • constrain the path in a
 8        ⟨η,u⟩ ← TCDRelProdSat(a[j'], n[i][j], r[j][j']);
 9        t[i][j'] ← Min(t[i][j'], ⟨η,u⟩);
10      endif
11    endfor
12  endfor
13  ⟨λ,t⟩ ← TCDSat(a, Normalize(t));
14  InsertUT(t);    CacheAdd_TCDRelProdSat(a, n, r, ⟨λ,t⟩);
15  return ⟨λ + μ,t⟩;
```

Fig. 5. Building TCD_p.

We divide the iteration into L phases according to the saturation scheme. The k^{th} phase begins only after $\langle\tau,t\rangle$ is saturated up to level $k-1$ and completes when it is saturated up to level k. An important idea is that every time we compute $\mathcal{N}_k(\langle\tau,t\rangle)$, we expect to keep the results saturated up to level k. These L-phase local fixpoint iterations execute bottom-up, until reaching the global fixpoint.

Figure 5 shows the pseudocode to compute TCD_p. This algorithm augments the transitive closure algorithm of [13] by computing distances between state pairs instead of a simple boolean reachability relation. MDD a encodes the set of states $\mathcal{S}_p$. Lines 7-10 and 16-18 in procedure $TCDSat$, and Lines 7-9 in procedure $TCDRelProdSat$ constrain all paths between pairs of states to be along states in the set encoded by MDD a. We assume that the MDDs encoding $\{\mathcal{N}_L,\ldots,\mathcal{N}_1\}$ have been computed and are globally available. As we use the QFI-reduction rule [14], $\mathcal{N}_k$ is an MDD with the root at level k.

The main procedure is a dual recursion between $TCDSat$ and $TCDRelProdSat$. $TCDSat$ computes the fixpoint in the k^{th} phase. In Line 17, it calls $TCDRelProdSat$ on lower levels to compute $\mathcal{N}_k(\langle\tau,t\rangle)$. $TCDRelProdSat$ computes $\mathcal{N}_k(\langle\tau,t\rangle)$ recursively and saturate the results at the end (Line 13) and thus returns a saturated result; this reflects the idea of aggressively computing local fixpoints on nodes as soon as they have been created. According to our experience [4], [6], [13], this scheme greatly speeds up convergence and reduces memory requirements in intermediate results for typical asynchronous systems.

While the saturation scheme exploits asynchronous event locality to speed up iterations, our algorithm does not impose any requirement on the system under verification. For systems where no natural asynchronous partition of the transition relation exists, such as fully synchronous systems, our algorithm is still applicable by letting $\mathcal{N}_L = \mathcal{N}$ and an empty $\mathcal{N}_k$ for $1 \le k < L$. In this case, the algorithm degrades to a stepwise procedure always operating from the top level, but still benefits from the efficiency of EV$^+$MDDs. In our experience, using an asynchronous partition and saturation, when possible, results in much faster runtime than using the monolithic next-state function and the stepwise procedure.

IV. Discussion

In this section, we discuss two extensions of our approach proposed above. First, we apply the idea in the above section to shortest witness generation for other properties in Section IV-A. Then, we tackle fairness in EG.

A. Shortest witness generation beyond EG

We extend the approach of Section III to shortest witness generation (SWG) for more general properties of the form $\mathsf{E}\psi$, where ψ is a path formula and $\mathsf{E}\psi$ does not necessarily have to be a CTL property. The resulting witnesses also constitute shortest counterexamples for $\mathsf{A}\neg\psi$.

Reviewing our TCD-based algorithm for shortest $\mathsf{EG}p$ witness generation, we can summarize the following steps:

1) Represent the length of a witness as an function, usually the sum of several *witness segments*. This is the objective of the minimization problem we need to solve and the minimal value is the shortest length of witnesses.
2) Encode the objective function with an EV$^+$MDD based on TCD, usually the sum of several EV$^+$MDDs, each of which corresponds to a witness segment.
3) Find a minimum solution using $MinState$ from EV$^+$MDD encoding the objective function. The solution can be several states "inducing" the shortest witness.
4) Build each witness segment, which is a shortest path between two states, and connect these segments sequentially to obtain a shortest witness.

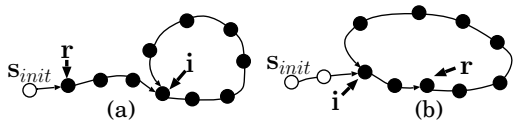

Fig. 6. A witness for $\mathsf{EF}(r \wedge \mathsf{EG}\neg s)$.

In SWG for EG, the objective function is Formula 1, the sum of stem and cycle. The central step is to find the minimal solution, knot $\mathbf{k}^*$, inducing the shortest witness. The extension of this approach to a complete framework able to handle all path formulas is non-trivial and beyond the scope of this paper. Instead, we present the basic idea in an informal way, by discussing the following two widely used properties.

• **Witnesses for** $\mathsf{E}(\mathsf{GF}p)$. We introduce function $CycleDist$ based on TCD:

$$CycleDist(\mathbf{i},\mathbf{j}) = \begin{cases} TCD(\mathbf{i},\mathbf{i}) & \text{if } \mathbf{i}=\mathbf{j} \\ TCD(\mathbf{i},\mathbf{j}) + TCD^{-1}(\mathbf{i},\mathbf{j}) & \text{otherwise,} \end{cases}$$

where $TCD = TCD_{true}$. Then, we need to find a state pair $(\mathbf{k},\mathbf{p})$, where $\mathbf{k}$ is the knot, which connects the stem and the cycle, and $\mathbf{p} \in \mathcal{S}_p$, which belongs to the cycle. Each witnesses consists of three segments: paths from $\mathbf{s}_{init}$ to $\mathbf{k}$, from $\mathbf{k}$ to $\mathbf{p}$, and from $\mathbf{p}$ to $\mathbf{k}$. The objective function for the minimization problem is

$$\min_{\mathbf{k}\in\mathcal{S},\mathbf{p}\in\mathcal{S}_p} \big(TCD^{stem}(\mathbf{k}) + CycleDist(\mathbf{k},\mathbf{p})\big), \qquad (2)$$

The minimal solution $(\mathbf{k}^*,\mathbf{p}^*)$ induces a shortest witness, consisting of three shortest witness segments.

• **Witnesses for** $\mathsf{E}[\mathsf{F}(r \wedge \mathsf{G}\neg s)]$. These are counterexamples to CTL properties of the form $\mathsf{AG}(r \rightarrow \mathsf{AF}s)$, which describe liveness: once a process issues a request (r), it will be eventually satisfied (s). Witnesses for $\mathsf{E}[\mathsf{F}(r \wedge \mathsf{G}\neg s)]$ reflect possible starvation in the system. For notational consistency with the previous section, let $p = \neg s$.

There are two types of witnesses for this property, as shown in Figure 6, where each circle denotes a state and solid black circles denote states in $\mathcal{S}_p$. We can solve these two cases separately and then find a global minimal result. In the first case, Figure 6(a), a witness consists of paths from $\mathbf{s}_{init}$ to a state $\mathbf{r} \in \mathcal{S}_r$ and from $\mathbf{r}$ to a knot $\mathbf{k}$, on a p-cycle, and such that p holds along the path between $\mathbf{r}$ and $\mathbf{k}$. In this case, there are three witness segments and the minimization objective is:

$$\min_{\mathbf{r},\mathbf{k}\in\mathcal{S}_p} \big(TCD^{stem}(\mathbf{r}) + TCD_p^{triv}(\mathbf{r},\mathbf{k}) + TCD_p^{cycle}(\mathbf{k})\big).$$

In the second case, Figure 6(b), the stem leads to a knot $\mathbf{k}$ on a p-cycle that contains a state $\mathbf{r} \in \mathcal{S}_r$. This case is similar to the SWG problem for $\mathsf{EGF}p$, except for replacing TCD with TCD_p in $CycleDist$, to constrain the cycles to $\mathcal{S}_p$. A shortest witness for this case can then be generated accordingly.

B. Shortest fair witness

In this section, we consider *Büchi fairness*, which can be specified with $n > 0$ sets of states $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$. A fair witness is a path leading to a *fair cycle*, which contains a state $\mathbf{i}_m \in \mathcal{F}_m$ for each $\mathcal{F}_m$. To simplify the discussion, we only explain how to generate shortest fair witness for $\mathsf{EG}true$, as the same idea can be extended to other properties. The complexity of this problem is proven to be NP-complete in [2].

We employ the idea in [8] by adding a *fairness flag* $\mathcal{S}_f$ as a submodel in TCD. $\mathcal{S}_f$ can be considered as a n-bit array, where i^{th} bit indicates whether the i^{th} fairness constraint has been fulfilled on a path. Let $\perp \in \mathcal{S}_f$ be the initial state where all bits are 0, and $\top \in \mathcal{S}_f$ be the state where all bits are 1, as all constraints are fulfilled. Define the operation $Set(\mathbf{f},m)$ to set the m^{th} bit to 1. The new TCD, denoted by TCD^f, can be expressed as an integer function on $(\mathbf{i},\mathbf{j},\mathbf{f})$, encoding the length of the shortest path that starts in $\mathbf{i}$, ends in $\mathbf{j}$, and satisfies the fair constraints indicated by $\mathbf{f}$. TCD^f can be build recursively by the following rule, using a similar algorithm as the one discussed above:

$$\begin{aligned} &\mathbf{j}\in\mathcal{N}(\mathbf{i}) \Rightarrow TCD^f(\mathbf{i},\mathbf{j},\perp) = 1 \\ &\wedge \min_{\mathbf{j}\in\mathcal{N}^{-1}(\mathbf{k})} (TCD^f(\mathbf{i},\mathbf{j},\mathbf{f})) = d \Rightarrow TCD^f(\mathbf{i},\mathbf{k},\mathbf{f}) = d+1 \\ &\wedge TCD^f(\mathbf{i},\mathbf{j},\mathbf{f}) = d \wedge \mathbf{j}\in\mathcal{F}_m \Rightarrow TCD^f(\mathbf{i},\mathbf{j},Set(\mathbf{f},m)) = d+1 \end{aligned}$$

Now the problem can be converted to witness generation without fair constraints. The minimization objective is:

$$\min_{\mathbf{k}\in\mathcal{S}} \big(TCD^{stem}(\mathbf{k}) + TCD^f(\mathbf{k},\mathbf{k},\top)\big).$$

The resulting witness can be mapped to the original system by filtering out n auxiliary steps setting fairness flag. This approach shares the same complexity with that in [8], and retains the benefits of using EV$^+$MDDs and saturation.

V. Experimental results

We implemented the proposed approach in SMART [15] and report experimental results running on an Intel Xeon 2.53GHz workstation with 36GB RAM under Linux 2.6.18. We also implemented the BFS-based algorithm of Figure 1 using MDD in SMART. We compare our results with those from the verification tool SAL [16]. Petri net models for SMART were converted to models in the SAL input language. The results from these algorithms are in the following columns:

• **"SMART-TCD"**: the TCD-based algorithm we propose. If it completes in the time limit, it returns a shortest witness with length L^*, used as an oracle for the other algorithms.

• **"SAL-BMC"**: Bounded model checker in SAL. We have two sets of runtimes, by setting the bound B to L^* and L^*-1 respectively, so that SAL-BMC tackles a satisfiable or unsatisfiable SAT problem, respectively.

• **"SMART-BFS"**: MDD-based witness generation implemented in SMART according to the algorithm in Figure 1. For the BFS-based algorithm, we run two sets of experiments. In Column "100 runs", we run the BFS-based algorithm 100 times and list the length of the shortest witness generated among these 100 runs, as well as the total runtime for the 100 runs. In subcolumn "L" and "time" respectively. In Column "runs till shortest", since we know L^* from the SMART-TCD, we run the BFS-based algorithm repeatedly until it generates one of the

shortest witnesses. The number of runs and runtimes required to generate the shortest witness using the BFS-based algorithm are listed in subcolumns "R" and "time", respectively. Also here we set a runtime limit of one hour. Since the BFS-based algorithm is randomized, the results in subcolumns "R" and "time" are the average over 100 experiments.

• **"SAL-WMC"**: BDD-based symbolic model checker in SAL. It generates a witness of length L without optimization.

The comparison metrics are runtime (columns "time", all measured in seconds) and length of the generated witness. Table I presents results on eight models, including mutual exclusion protocols (*peterson* and *bakery* in [17]), leader election protocol (*leader*), the dining philosopher problem (*phil*), a closed queue network (*cqn*), an arbiter protocol (*arbiter*), a factory automation model (*kanban*), and the two *robin* and *slot* protocols [18]. The sizes of the state spaces of these models are parameterized by an integer N. The first three columns list the model names, the parameters, and the sizes of state spaces.

A bounded model checker can find the shortest witness using binary search; this requires running a SAT solver $O(\lceil log_2 L^* \rceil)$ times, to both generate a shortest counterexample and prove that no shorter counterexamples exist. Thus, the sum of the runtimes for $B = L^*$ and $B = L^* - 1$ is a reasonable lower bound for the runtime required by SAL-BMC to find a shortest witness. The results in Column "SMART-TCD" and "SAL-BMC" shows that SAL-BMC achieves obvious speedup over SMART-TCD only on *kanban*, but performs much worse in *robin*, *slot* and *cqn*. Even provided with L^* as the bound, SAL-BMC still requires much more time to find the witness in these three models. These results demonstrate the efficiency of our approach.

SAL-WMC and SMART-BFS are based on the same idea, but use different data structures, i.e., BDDs vs. MDDs. Neither of them can guarantee shortest witnesses. However, SMART-BFS runs much faster than SAL-WMC, due to the efficiency of our MDD library and of our encoding of next-state functions. It is not surprising that SMART-BFS runs orders of magnitude faster than SMART-TCD because computing TCD is much more expensive than the image computations in SMART-BFS. On the other hand, SMART-TCD generates much shorter witness for *slot*, *arbiter* and *cqn* than SMART-BFS. Thanks to EV$^+$MDD and saturation, SMART-TCD completes on complex models with more than 10^{10} states and, on *cqn* and *phils*, it runs even faster than SAL-WMC, which does not attempt to minimize the witness length.

For *cqn* and *arbiter*, SMART-TCD generates much shorter witnesses than SAL, while SMART-BFS fails to find a shortest witness within the time limit. Figure 7 illustrates how the runtime increases and the shortest length of witnesses found decreases as the BFS-based algorithm runs repeatedly and cumulatively in model *cqn*20. Similar results can be observed in *arbiter*, but we do not show it here because of space limitations. The x-axis (in logarithmic scale) indicates the total number of runs, the solid line (associated with the left y-axis) shows the total runtime, and the dotted line (associated with the right y-axis) shows the shortest length of witnesses found. For comparison, the thin solid line and the dotted line mark the runtime SMART-TCD and L^*, respectively. We can see that runtime grows almost linearly, and many runs (recall that the x-axis is in logarithmic scale) are needed to find a short witness. Within the given runtime, the SMART-BFS produces much longer witnesses than the SMART-TCD. This is analogous to simulation-based verification, which, while it provides good coverage for simple designs, requires unacceptable runtimes to reach corner cases in complex designs. SMART-BFS randomly chooses the next step at each iteration, just as in unguided simulation. If there are only a few witnesses, as in *phils* and *kanban*, SMART-BFS can find a shortest witness in few runs with high probability, although, even in these cases, it cannot prove that there is no shorter witness without exhaustively searching all possible witnesses. If there are many witnesses, SMART-BFS might instead only be able to generate very long witnesses even after a long runtime, as Figure 7 illustrates. In this case the SMART-TCD becomes a better choice to generate a short witness, indeed a guaranteed shortest witness.

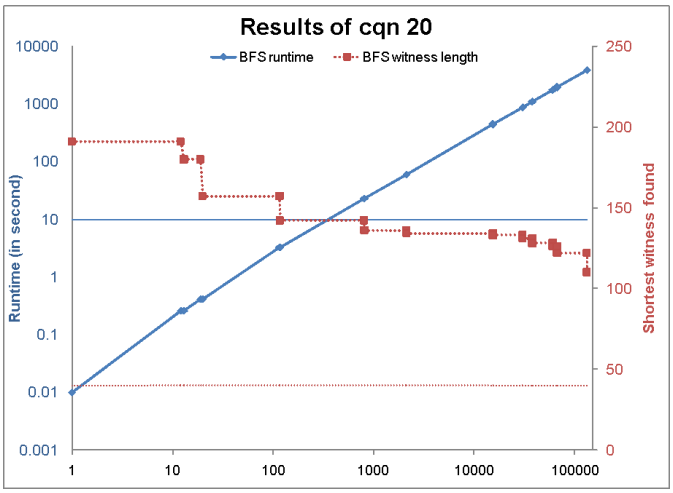

Fig. 7. Runtime and witness length of the BFS-based algorithm on *cqn*.

VI. Conclusion and Future Work

We presented a saturation-based algorithm for shortest EG witness generation. We proposed a symbolic techniques using EV$^+$MDDs to compute the Transitive Closure with Distance (TCD), which compactly represents distances between each pair of states. Then, the shortest EG witness can be identified symbolically. We also extended this approach to tackle shortest witness generation for other properties and shortest fair witness generation.

Computing TCD is the bottleneck in our approach. Techniques to speed up this computation should be investigated, including dynamic variable ordering. Coupled with EV$^+$MDDs, the transitive closure provides an elegant way of analyzing quantitative properties of traces in complex asynchronous systems, such as probabilistic model checking, which we intend to investigate in future work.

Model	N	SS	SMART-TCD		SAL-BMC		SAL-WMC		SMART-BFS			
									100 runs		runs till shortest	
			L^*	time	time $B=L^*$	time $B=L^*-1$	L	time	L	time	R	time
$kanban$	6	1.12×10^7	3	7.93	0.0	0.1	18	10.37	3	< 0.01	2.96	< 0.01
	8	1.33×10^8	3	67.86	0.1	0.1	10	16.85	3	< 0.01	3.00	< 0.01
	10	1.00×10^9	3	441.28	0.1	0.1	10	62.09	3	< 0.01	2.97	< 0.01
$leader$	3	8.49×10^2	15	0.47	0.1	8.56	15	0.03	15	0.03	3.36	< 0.01
	4	1.15×10^4	20	34.60	0.47	1017.33	59	0.80	20	0.18	11.75	0.03
	5	1.50×10^5	25	4746.63	4.14	TO	149	14.30	25	0.52	100.67	0.58
$phils$	10	1.86×10^6	4	0.05	0.08	0.04	38	0.32	4	0.04	21.91	0.01
	20	3.46×10^{12}	4	0.26	0.05	0.25	47	42.07	4	0.06	24.90	0.02
	100	4.96×10^{62}	4	45.58	0.57	35.98	–	TO	4	0.13	26.85	0.06
$robin$	10	2.30×10^4	40	0.07	5.35	TO	43	0.35	40	0.04	1.08	< 0.01
	20	4.71×10^7	80	0.30	321.24	TO	83	8.42	80	0.27	1.06	0.01
	30	7.24×10^{10}	120	0.78	3957.47	TO	120	152.07	120	0.65	1.04	0.03
$slot$	5	5.38×10^4	17	2.26	11.95	5.18	131	0.17	21	0.10	3620.16	4.16
	6	5.75×10^5	20	11.08	0.84	142.49	182	0.78	25	0.24	93411.96	243.99
	7	6.22×10^6	23	46.00	4197.17	611.84	483	2.43	46	0.51	–	TO
$arbiter$	10	2.04×10^4	10	0.26	1.27	2.45	37	0.1	20	0.05	–	TO
	15	9.83×10^5	15	19.31	33.01	49.71	74	0.44	39	0.11	–	TO
	20	4.19×10^7	20	2625.28	1597.63	1025.43	107	0.91	58	0.32	–	TO
cqn	20	1.93×10^{11}	40	9.76	5682.55	2542.93	–	TO	150	2.51	–	TO
	30	1.66×10^{17}	60	183.20	TO	TO	–	TO	322	10.17	–	TO
	40	1.51×10^{23}	80	3322.41	TO	TO	–	TO	625	25.97	–	TO
$peterson$	2	2.28×10^2	11	0.11	0.02	0.08	12	0.06	12	< 0.01	866.04	0.05
	3	1.47×10^4	24	477.29	7.84	244.54	37	0.61	37	0.32	–	TO
$bakery$	2	1.11×10^3	11	0.27	0.04	0.10	11	0.1	22	0.04	1100.66	0.10
	3	1.39×10^5	–	TO	N/A	N/A	161	2.71	65	2.21	N/A	N/A

TABLE I
RESULTS FOR EG WITNESS GENERATION.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[2] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *32nd Design Automation Conference (DAC 95)*, 1995, pp. 427–432.

[3] V. Schuppan and A. Biere, "Shortest counterexamples for symbolic model checking of LTL with past," in *Proc. TACAS*, ser. Lecture Notes in Computer Science, vol. 3440. Springer, 2005, pp. 493–509.

[4] G. Ciardo and R. Siminiceanu, "Using edge-valued decision diagrams for symbolic generation of shortest paths," in *Proc. FMCAD*, ser. LNCS 2517. Springer, 2002, pp. 256–273.

[5] S. Kashyap and V. K. Garg, "Producing short counterexamples using "crucial events"," in *Proc. CAV*, ser. CAV '08. Springer, 2008, pp. 491–503.

[6] G. Ciardo, R. Marmorstein, and R. Siminiceanu, "The saturation algorithm for symbolic state space exploration," *Software Tools for Technology Transfer*, vol. 8, no. 1, pp. 4–25, 2006.

[7] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Multi-valued decision diagrams: theory and applications," *Multiple-Valued Logic*, vol. 4, no. 1–2, pp. 9–62, 1998.

[8] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *Software Tools for Technology Transfer*, vol. 5, no. 2, pp. 185–204, March 2004.

[9] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue, "Heuristic-guided counterexample search in FLAVERS," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, ser. SIGSOFT '04/FSE-12. ACM, 2004, pp. 201–210.

[10] Y.-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," in *Proceedings of the 29th Conference on Design Automation*. IEEE Computer Society Press, Jun. 1992, pp. 608–613.

[11] P. Roux and R. Siminiceanu, "Model Checking with Edge-valued Decision Diagrams," in *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*. NASA, April 2010, pp. 222–226.

[12] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*. Springer, 1999, pp. 193–207.

[13] Y. Zhao and G. Ciardo, "Symbolic CTL model checking of asynchronous systems using constrained saturation," in *Proc. ATVA*, ser. LNCS 5799. Springer, 2009, pp. 368–381.

[14] M. Wan and G. Ciardo, "Symbolic state-space generation of asynchronous systems using extensible decision diagrams," in *Proc. SOFSEM*, ser. LNCS 5404. Springer, 2009, pp. 582–594.

[15] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu, "Logical and stochastic modeling with SMART," *Perf. Eval.*, vol. 63, pp. 578–608, 2006.

[16] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Proc. CAV*, ser. Lecture Notes in Computer Science, vol. 3114. Springer, Jul. 2004, pp. 496–500.

[17] R. Pelánek, "BEEM: benchmarks for explicit model checkers," in *Proceedings of the 14th international SPIN conference on Model checking software*. Springer, 2007, pp. 263–267.

[18] G. Ciardo *et al.*, "SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual," available at http://www.cs.ucr.edu/∼ciardo/SMART/.