





# Generation of Minimum Tree-Like Witnesses for Existential CTL

Chuan Jiang<sup>(✉)</sup>  and Gianfranco Ciardo 

Department of Computer Science,  
Iowa State University, Ames, IA 50011, USA  
{cjiang,ciardo}@iastate.edu



**Abstract.** An advantage of model checking is its ability to generate witnesses or counterexamples. Approaches exist to generate small or minimum witnesses for simple unnested formulas, but no existing method guarantees minimality for general nested ones. Here, we give a definition of witness size, use edge-valued decision diagrams to recursively compute the minimum witness size for each subformula, and describe a general approach to build minimum tree-like witnesses for existential CTL. Experimental results show that for some models, our approach is able to generate minimum witnesses while the traditional approach is not.

## 1 Introduction

Model checking is an automated technique to rigorously establish the correctness of a system by exploring its computation graph, explicitly or symbolically. Instead of merely answering “yes” or “no”, model checkers may be able to return a witness or counterexample to verify satisfaction or violation of a specification. Since witnesses and counterexamples provide important debugging information and may be inspected by engineers, smaller ones are always preferable.

Computation Tree Logic (CTL) is widely used to express temporal properties due to its simple yet expressive semantics. Although much work has been published on witness or counterexample generation [8, 11, 14], to the best of our knowledge, no existing method guarantees their minimality for a general CTL formula with nested temporal quantifiers. Clarke et al. [7] showed that the general form of a counterexample to a universal CTL formula is tree-like; of course, for CTL, counterexample generation for a universal formula can be converted to witness generation for an existential formula, thus we choose to limit our discussion to witness generation for the existential fragment of CTL. The use of backward exploration to verify **EX**, **EF**, and **EU** properties inherently guarantees minimality of their linear witnesses, while a minimum lasso-shaped **EG** witness can be generated by computing transitive closures, for example using the saturation algorithm [17]. However, these approaches do not extend to general tree-like witnesses, i.e., local minimality does not imply global minimality.

By recursively computing local fixpoints, the saturation algorithm [3] has clear advantages over traditional symbolic breadth-first approaches for state-space generation. It has also been applied to the computation of minimum **EF** [4]

and **EG** [17] witnesses. Here, we extend these ideas into a global approach to build minimum witnesses for arbitrary existential CTL formulas.

Our paper is organized as follows. Section 2 summarizes background on CTL, decision diagrams, and symbolic CTL model checking. Section 3 defines the witness size and formalizes the computation of its minimum. Section 4 proposes saturation-based algorithms to symbolically encode minimum witness sizes for each existential temporal operator, needed to obtain an overall minimum witness size. Section 5 describes how to generate a witness from the computed witness size functions. Section 6 presents experimental results, Sect. 7 comments on our definition of witness size, and Sect. 8 concludes and outlines future work.

## 2 Background

We denote sets using calligraphic letters (e.g.,  $\mathcal{A}$ ,  $\mathcal{B}$ ,  $\mathcal{C}$ ), except for the booleans  $\mathbb{B} = \{0, 1\}$ , the natural numbers  $\mathbb{N} = \{0, 1, 2, \dots\}$ , and  $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ .

### 2.1 Kripke Structures, CTL, and Witnesses

A Kripke structure is a tuple  $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$ , where  $\mathcal{S}$  is the state space,  $\mathcal{S}_{init} \subseteq \mathcal{S}$  are the initial states,  $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$  is the next-state function,  $\mathcal{A}$  is a set of atomic propositions, and  $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{A}}$  is a labeling that gives the atomic propositions holding in each state (subject to  $true \in \mathcal{A}$  holding in every state).

We assume  $\mathcal{S}$  to be the product  $\mathcal{S}_1 \times \dots \times \mathcal{S}_L$  of  $L$  finite state spaces, i.e., each *global* state  $\mathbf{i} \in \mathcal{S}$  is a tuple  $(i_1, \dots, i_L)$ , where  $i_k \in \mathcal{S}_k$  is the *local* state for the  $k^{\text{th}}$  submodel. We also assume  $\mathcal{N}$  to be disjointly partitioned according to a set  $\mathcal{E}$  of *asynchronous* events, i.e.,  $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$  and, for each  $e \in \mathcal{E}$ ,  $\mathcal{N}_e : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ .  $\mathcal{N}_e(\mathbf{i})$  contains the states that can be nondeterministically reached in one step when event  $e$  occurs in state  $\mathbf{i}$ . Correspondingly, we let  $\mathcal{N}_e^{-1}$  denote the previous-state function, i.e.,  $\mathcal{N}_e^{-1}(\mathbf{j}) = \{\mathbf{i} : \mathbf{j} \in \mathcal{N}_e(\mathbf{i})\}$ , the set of states that can reach  $\mathbf{j}$  in one step through the occurrence of event  $e$ , and we let  $\mathcal{N}^{-1} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e^{-1}$ .

Let  $\mathcal{P}(\mathbf{i})$  be the set of paths starting at  $\mathbf{i} \in \mathcal{S}$ , i.e., finite sequences  $(\mathbf{i}_0, \mathbf{i}_1, \dots, \mathbf{i}_n)$  for  $n \geq 0$ , or infinite sequences  $(\mathbf{i}_0, \mathbf{i}_1, \dots)$ , where  $\mathbf{i}_0 = \mathbf{i}$  and  $\mathbf{i}_m \in \mathcal{N}(\mathbf{i}_{m-1})$  for all applicable  $m$ . Let  $\mathcal{C}(\mathbf{i}) \subseteq \mathcal{P}(\mathbf{i})$  be the set of cycles starting at  $\mathbf{i}$ , i.e., finite paths  $(\mathbf{i}_0, \mathbf{i}_1, \dots, \mathbf{i}_n)$  with  $n > 0$  and  $\mathbf{i}_0 = \mathbf{i}_n = \mathbf{i}$ .

We consider ECTL, the existential fragment of the temporal logic CTL [6], where formulas have syntax ( $\phi$  and  $\rho$  are formulas,  $a$  is an atomic proposition):

$$\phi ::= a \mid \neg a \mid \phi \wedge \rho \mid \phi \vee \rho \mid \mathbf{E}\phi \mid \mathbf{E}\phi\mathbf{U}\rho \mid \mathbf{E}\mathbf{G}\phi,$$

and the conditions for state  $\mathbf{i}$  to satisfy formula  $\phi$ , written  $\mathbf{i} \models \phi$ , are as follows:

$$\begin{aligned} \mathbf{i} \models a &\iff a \in \mathcal{L}(\mathbf{i}) \\ \mathbf{i} \models \neg a &\iff a \notin \mathcal{L}(\mathbf{i}) \\ \mathbf{i} \models \phi \wedge \rho &\iff \mathbf{i} \models \phi \text{ and } \mathbf{i} \models \rho \\ \mathbf{i} \models \phi \vee \rho &\iff \mathbf{i} \models \phi \text{ or } \mathbf{i} \models \rho \\ \mathbf{i} \models \mathbf{E}\phi &\iff \exists (\mathbf{i}_0, \mathbf{i}_1) \in \mathcal{P}(\mathbf{i}), \mathbf{i}_1 \models \phi \\ \mathbf{i} \models \mathbf{E}\phi\mathbf{U}\rho &\iff \exists (\mathbf{i}_0, \mathbf{i}_1, \dots, \mathbf{i}_n) \in \mathcal{P}(\mathbf{i}), n \geq 0, \mathbf{i}_n \models \rho \wedge \forall m \in \{0, \dots, n-1\}, \mathbf{i}_m \models \phi \\ \mathbf{i} \models \mathbf{E}\mathbf{G}\phi &\iff \exists (\mathbf{i}_0, \mathbf{i}_1, \dots) \in \mathcal{P}(\mathbf{i}), \forall m \geq 0, \mathbf{i}_m \models \phi \end{aligned}$$

(formula  $\mathbf{E}\mathbf{F}\phi$  is just a shorthand for  $\mathbf{E}true\mathbf{U}\phi$ , so we do not discuss it separately).

Since the state space  $\mathcal{S}$  is finite, all infinite paths contain a cycle. Thus, path  $(\mathbf{i}_0, \mathbf{i}_1, \dots)$  demonstrating  $\mathbf{i} \models \mathbf{EG}\phi$  must have a finite prefix  $(\mathbf{i}_0, \dots, \mathbf{i}_m, \dots, \mathbf{i}_n)$ , for some  $m \geq 0$  and  $n > m$ , where  $\mathbf{i}_m = \mathbf{i}_n$ , that is, it is a “lasso” formed by merging (on state  $\mathbf{i}_m$ ) a possibly empty “handle”  $(\mathbf{i}_0, \dots, \mathbf{i}_m)$  and a “cycle”  $(\mathbf{i}_m, \dots, \mathbf{i}_n)$ .

We focus on witness generation, i.e., the computation of “tree-like” subgraphs demonstrating how a state satisfies an ECTL formula. This also serves to generate counterexamples for ACTL, the universal fragment of CTL, since a counterexample to  $\mathbf{AX}\phi$ ,  $\mathbf{AG}\phi$ , or  $\mathbf{A}[\phi\mathbf{U}\rho]$  is a witness to  $\mathbf{EX}(\neg\phi)$ ,  $\mathbf{EF}(\neg\phi)$ , or  $\mathbf{E}[\neg\rho\mathbf{U}(\neg\phi \wedge \neg\rho)] \vee \mathbf{EG}(\neg\rho)$ , respectively (where the negation  $\neg$  can be “pushed down” to atomic propositions), i.e., the negation of an ACTL formula is an ECTL formula.

## 2.2 Decision Diagrams

We encode sets and relations symbolically with (ordered) *multiway decision diagrams* (MDDs) [10]. An  $L$ -level MDD over  $\mathcal{S} = \mathcal{S}_1 \times \dots \times \mathcal{S}_L$  is an acyclic directed edge-labeled level graph with *terminal* nodes 0 and 1, at level 0, while each *non-terminal* node  $p$  is at some level  $p.lvl = k \in \{1, \dots, L\}$ , and, for  $i_k \in \mathcal{S}_k$ , has an outgoing edge labeled with  $i_k$  and pointing to a child  $p[i_k]$  at level  $p[i_k].lvl < k$ .

MDD node  $p$  at level  $k$  encodes function  $f_p : \mathcal{S} \rightarrow \mathbb{B}$ , recursively defined by  $f_p(i_1, \dots, i_L) = f_{p[i_k]}(i_1, \dots, i_L)$ , with base case  $f_p(i_1, \dots, i_L) = p$  when  $k = 0$ . Interpreting  $f_p$  as an indicator function,  $p$  encodes set  $\mathcal{X}_p = \{\mathbf{i} : f_p(\mathbf{i}) = 1\} \subseteq \mathcal{S}$ . To encode relations over  $\mathcal{S}$ , we use  $2L$ -level MDDs over  $(\mathcal{S}_1 \times \mathcal{S}_1) \times \dots \times (\mathcal{S}_L \times \mathcal{S}_L)$ , where the first set in each pair corresponds to a “from”, or “*unprimed*”, local state and the second set corresponds to a “to”, or “*primed*”, local state.

We use instead (ordered) *additive edge-valued MDDs*, (EV<sup>+</sup>MDDs) [4] to encode partial integer-valued functions. An EV<sup>+</sup>MDD is an acyclic directed edge-labeled and edge-valued level graph with terminal node  $\Omega$ , at level 0, while each nonterminal node  $p$  is at some level  $p.lvl = k \in \{1, \dots, L\}$ , and, for  $i_k \in \mathcal{S}_k$ , has an outgoing edge with label  $i_k$ , pointing to a child  $p[i_k].c$  at a level  $p[i_k].c.lvl < k$ , and value  $p[i_k].v \in \mathbb{N}_\infty$ . We write  $p[i_k] = \langle p[i_k].v, p[i_k].c \rangle$ .

EV<sup>+</sup>MDD node  $p$  at level  $k$  encodes function  $f_p : \mathcal{S} \rightarrow \mathbb{N}_\infty$  recursively defined by  $f_p(i_1, \dots, i_L) = p[i_k].v + f_{p[i_k].c}(i_1, \dots, i_L)$ , with base case  $f_\Omega(i_1, \dots, i_L) = 0$ .

For efficiency, we restrict ourselves to *canonical* forms of decision diagrams, where each function that can be encoded by a given class of decision diagrams has a unique representation in that class. All such forms forbid *duplicate nodes*: if  $p.lvl = q.lvl = k > 0$  and  $\forall i_k \in \mathcal{S}_k, p[i_k] = q[i_k]$ , then  $p = q$ . For ease of exposition, we only consider the *quasi-reduced* form in this paper, achieved by forbidding *skipped levels*: all roots (nodes without parent nodes) are at level  $L$  and, if  $p.lvl = k$ , then all  $p$ ’s children are at level  $k - 1$ . For EV<sup>+</sup>MDDs, in addition, we require *normalized nodes*: each nonterminal node must have at least one edge with value 0 and all edges with value  $\infty$  must point to  $\Omega$ . This means that the minimum value of the function encoded by any node is 0, but we can encode any partial function  $g : \mathcal{S} \rightarrow \mathbb{N}_\infty$  with a “root edge”  $\langle \sigma, p \rangle$ , where  $\sigma$  is the minimum value assumed by  $g$ , while  $p$  at level  $L$  satisfies  $f_p = g - \sigma$ .

---

```

Min(EV+MDD  $\langle \alpha, p \rangle$ , EV+MDD  $\langle \beta, q \rangle$ )
1: if  $\alpha = \infty$  or  $\beta = \infty$  then return  $\langle \infty, \Omega \rangle$ 
2:  $k \leftarrow p.lvl$  ▷ we assume quasi-reduced rule, thus  $p.lvl = q.lvl$ 
3: if  $k = 0$  then return  $\langle \min\{\alpha, \beta\}, \Omega \rangle$  ▷ terminal case,  $p = q = \Omega$ 
4: if MinGet( $p, q, \alpha - \beta, u$ ) then return  $\langle \min\{\alpha, \beta\}, u \rangle$  ▷  $Min(p, q, \alpha - \beta) = \langle \min\{\alpha, \beta\}, u \rangle$ 
5:  $u \leftarrow$  EVMDNode( $k$ ) ▷ create a new EV+MDD node at level  $k$ 
6: for each  $i \in S_k$  do ▷ recursively compute the children of the result node
7:    $u[i] \leftarrow$  Min( $\langle \alpha + p[i].v, p[i].c \rangle, \langle \beta + q[i].v, q[i].c \rangle$ )
8:  $\langle \mu, u \rangle \leftarrow$  Normalize( $u$ ) ▷ subtract  $\mu$  to  $u$ 's edge values so that the minimum is 0
9: ▷ ...unless they are all  $\infty$ , in which case  $\langle \mu, u \rangle = \langle \infty, \Omega \rangle$ 
10: MinPut( $p, q, \alpha - \beta, u$ ) ▷ memoize the result
11: return  $\langle \mu, u \rangle$ 
    
```

---

**Fig. 1.** Algorithm to compute the element-wise minimum of two functions.

However, our algorithms are actually implemented using the more efficient *fully-identity-reduced* form for  $2L$ -level MDDs and EV<sup>+</sup>MDDs (indicated MDD2 and EV<sup>+</sup>MDD2 in our algorithms, respectively) [5]. This form allows us to exploit independence of events from local states: given  $e \in \mathcal{E}$ , let  $Top(e) = k$  if  $e$  affects or depends on the  $k^{\text{th}}$  local state but not the  $l^{\text{th}}$  one, for any  $l > k$ . In the following, we then define  $\mathcal{N}_k = \bigcup_{e: Top(e)=k} \mathcal{N}_e$ .

Procedure *Min* in Fig. 1 shows the classic recursive manipulation of decision diagrams. Given functions  $f, g : \mathcal{S} \rightarrow \mathbb{N}_\infty$ , let  $Min_{f,g} : \mathcal{S} \rightarrow \mathbb{N}_\infty$  be their element-wise minimum: for  $\mathbf{i} \in \mathcal{S}$ ,  $Min_{f,g}(\mathbf{i}) = \min\{f(\mathbf{i}), g(\mathbf{i})\}$ . Given two EV<sup>+</sup>MDDs  $\langle \alpha, p \rangle$  and  $\langle \beta, q \rangle$  encoding  $f$  and  $g$ , procedure *Min* returns the EV<sup>+</sup>MDD encoding  $Min_{f,g}$ . As the EV<sup>+</sup>MDDs are quasi-reduced,  $p.lvl = q.lvl$  unless  $\alpha = \infty$  or  $\beta = \infty$ .

Procedure *Normalize* (line 8) normalizes a node  $u$  by subtracting the minimum edge value  $\mu$  from all its edge values, so that at least one is 0, stores the normalized  $u$  in the unique table (if not already there), and returns  $\langle \mu, u \rangle$ .

Throughout this paper, procedures *XxxGet* (line 4) or *XxxPut* (line 10) are queries to or insertions into compute tables (or “caches”), commonly used in decision diagrams operations to avoid re-computation. The structure of the hash key and returned value may of course depend on the specific operation *Xxx*.

### 2.3 Symbolic CTL Algorithms

McMillan proposed symbolic CTL model checking based on binary decision diagrams (BDDs) [12]. Given the BDDs encoding the set of states satisfying  $\phi$  and  $\rho$ , algorithms to compute the BDD encoding the set of states satisfying  $\mathbf{EX}\phi$ ,  $\mathbf{E}\phi\mathbf{U}\rho$ , and  $\mathbf{E}G\phi$  suffice, since all CTL formulas can be expressed using these three CTL operators, plus the standard logical operations of negation, conjunction, and disjunction. Using MDDs instead of BDDs is a relatively obvious extension.

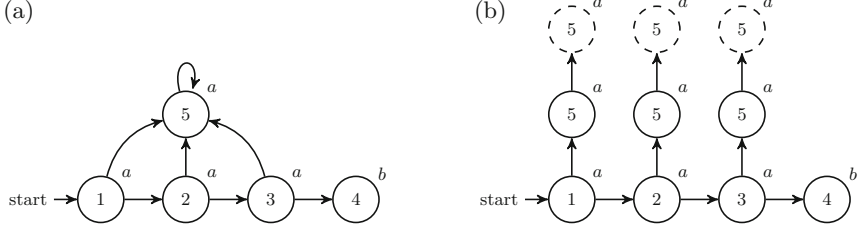
Clarke et al. [8] proposed the first symbolic approach to CTL witness generation. Considering first unnested CTL formulas, a witness for  $\mathbf{EX}a$  can be generated by one image computation, and is by definition minimum since all witnesses have

size two. Using a symbolic breadth-first search, witness generation for  $\mathbf{EaUb}$  also guarantees minimality, while minimality is more difficult to satisfy for  $\mathbf{EGa}$ , where a witness is a path from an initial state to a cycle, such that all states along that path and on the cycle satisfy  $a$ . In other words, a state  $\mathbf{i}$  satisfying  $\mathbf{EGa}$  must have a successor also satisfying  $\mathbf{EGa}$ ; thus, we can incrementally build a path of states satisfying  $\mathbf{EGa}$ , which must finally lead to a state already on the path, closing the cycle and resulting in a witness. A witness generation algorithm for (weakly fair)  $\mathbf{EG}$  was proposed in [8] based on this idea. Since  $\mathbf{i}$  might have multiple successors satisfying  $\mathbf{EGa}$ , the algorithm is nondeterministic and the size of the witness depends on the state chosen at each step. While the algorithm uses a symbolic encoding, the approach is largely explicit, as it follows a single specific path. Decision diagrams help by efficiently encoding all states satisfying  $\mathbf{EG}$ , but offer no help at all when deciding which of the states in  $\mathcal{N}(\mathbf{i})$  satisfying  $\mathbf{EG}$  should be chosen next, to continue the path from  $\mathbf{i}$ .

Witness generation for arbitrarily nested CTL formulas is much harder. Of course, we cannot exhibit witnesses for universal formulas, only counterexamples, thus the presence of both existential and universal (non-negated) quantifiers in a CTL formula  $\phi$  means that we can neither provide a witness (in case  $\phi$  holds) nor a counterexample (in case  $\phi$  does not hold). The most general approach to date is by Clarke et al. [7] for general (nested) ACTL formulas, which proposed algorithms to generate tree-like counterexamples, or witnesses for general (nested) ECTL formulas. However, their work did not address minimality.

### 3 Defining the Minimum Witness Size

We focus on the generation of *minimum* witnesses for general (nested) ECTL formulas. As discussed by Clarke et al. [7], these witnesses are finite tree-like Kripke structures and complete for ECTL. To discuss their size, we unfold these witnesses, i.e., the same state may appear multiple times and each appearance contributes to the count defining the size of the witness. For example, consider the (portion of a) Kripke structure shown in Fig. 2(a), satisfying formula  $\mathbf{E(EGa)Ub}$  (state are identified by numbers and the atomic propositions holding in each state are listed close to it). An unfolded tree-like witness for this formula is shown in Fig. 2(b), where state 5's self-loop is repeated three times, once for each of the states 1, 2, and 3, since we need to show that each of them satisfies  $\mathbf{EGa}$  (for clarity, a cycle is represented as a linear path along which the first and the last states are the same; dashed nodes represent the states closing cycles). Another way to think of this witness is that the first states of paths  $\llbracket 1, 5, \bar{5} \rrbracket$ ,  $\llbracket 2, 5, \bar{5} \rrbracket$ , and  $\llbracket 3, 5, \bar{5} \rrbracket$ , each satisfying the inner formula  $\phi' = \mathbf{EGa}$ , are “glued” onto the first three states of path  $\llbracket 1, 2, 3, 4 \rrbracket$ , satisfying the outermost formula  $\mathbf{E}\phi'\mathbf{U}\phi''$ , as is the first (and only) state of path  $\llbracket 4 \rrbracket$ , satisfying the (atomic) inner formula  $\phi'' = b$ . We write  $\bar{5}$  for the last state of the  $\mathbf{EG}$  witnesses to stress that state 5 does not need to have its witness repeated, since it is just closing the cycle. We define the *size* of a witness as the number of nodes in the resulting tree-like graph. Thus, a witness for  $a$  is path  $\llbracket 1 \rrbracket$ , of size 1, a witness for  $\mathbf{EXa}$



**Fig. 2.** A Kripke structure satisfying  $\mathbf{E}(\mathbf{E}G a)\mathbf{U}b$  and its tree-like witness.

is path  $\llbracket 1, 2 \rrbracket$ , of size 2, a witness for  $\mathbf{E}a\mathbf{U}b$  is path  $\llbracket 1, 2, 3, 4 \rrbracket$ , of size 4, and a witness to  $\mathbf{E}(\mathbf{E}G a)\mathbf{U}b$  is the three-like graph  $\llbracket \llbracket 1, 5, \bar{5} \rrbracket, \llbracket 2, 5, \bar{5} \rrbracket, \llbracket 3, 5, \bar{5} \rrbracket, 4 \rrbracket$ , of size 10. For conjunction, we need additional path notation: a witness for  $\mathbf{E}X a \wedge \mathbf{E}a\mathbf{U}b$  is a tree-like graph  $\llbracket \llbracket 1, 5 \rrbracket \diamond \llbracket 1, 2, 3, 4 \rrbracket \rrbracket$ , of size 5, where the separator  $\diamond$  indicates that the tree-like graphs to its left and its right are to be merged on their root.

We recursively define function  $\pi_\phi : \mathcal{S} \rightarrow \mathbb{N}_\infty$  describing the minimum witness size for an ECTL formula  $\phi$  starting from a state  $\mathbf{i} \in \mathcal{S}$  as follows:

$$\begin{aligned}
 \pi_a(\mathbf{i}) &= \text{if } \mathbf{i} \models a: 1, \text{ else: } \infty \\
 \pi_{\neg a}(\mathbf{i}) &= \text{if } \mathbf{i} \models a: \infty, \text{ else: } 1 \\
 \pi_{\phi \wedge \rho}(\mathbf{i}) &= \pi_\phi(\mathbf{i}) + \pi_\rho(\mathbf{i}) - 1 \quad (\text{the “-1” avoids double-counting state } \mathbf{i}) \\
 \pi_{\phi \vee \rho}(\mathbf{i}) &= \min\{\pi_\phi(\mathbf{i}), \pi_\rho(\mathbf{i})\} \\
 \pi_{\mathbf{E}X\phi}(\mathbf{i}) &= \min\{\pi_\phi(\mathbf{j}) : \forall \mathbf{j} \in \mathcal{N}(\mathbf{i})\} + 1 \\
 \pi_{\mathbf{E}\phi\mathbf{U}\rho}(\mathbf{i}) &= \text{if } \mathbf{i} \models \mathbf{E}\phi\mathbf{U}\rho: \min\{\pi_\rho(\mathbf{i}), \pi_\phi(\mathbf{i}) + \min\{\pi_{\mathbf{E}\phi\mathbf{U}\rho}(\mathbf{j}) : \forall \mathbf{j} \in \mathcal{N}(\mathbf{i})\}\}, \text{ else: } \infty \\
 \pi_{\mathbf{E}G\phi}(\mathbf{i}) &= \text{if } \mathbf{i} \models \mathbf{E}G\phi: \min\{\chi_\phi(\mathbf{i}), \pi_\phi(\mathbf{i}) + \min\{\pi_{\mathbf{E}G\phi}(\mathbf{j}) : \forall \mathbf{j} \in \mathcal{N}(\mathbf{i})\}\}, \text{ else: } \infty \\
 \chi_\phi(\mathbf{i}) &= \text{if } \mathcal{C}(\mathbf{i}) \neq \emptyset: \min\{\sum_{i=1}^n \pi_\phi(\mathbf{i}_i) : \forall (\mathbf{i}_0, \mathbf{i}_1, \dots, \mathbf{i}_n) \in \mathcal{C}(\mathbf{i})\} + 1, \text{ else: } \infty
 \end{aligned}$$

where  $\mathcal{C}(\mathbf{i})$  is the set of cycles starting at  $\mathbf{i}$ , and  $\chi_\phi(\mathbf{i})$  is the minimum witness size among cycles satisfying  $\mathbf{E}G\phi$  and starting at  $\mathbf{i}$ . In the sum for  $\chi_\phi(\mathbf{i})$ , we exclude  $\pi_\phi(\mathbf{i}_0)$  and add 1 because state  $\mathbf{i} = \mathbf{i}_0 = \mathbf{i}_n$  starting and ending the cycle appears twice, but we should not count the witness for  $\mathbf{i} \models \phi$  twice.

## 4 Computing the Minimum Witness Size

The first and most complex step to generate a minimum tree-like witness for an arbitrary ECTL formula  $\phi^*$  is to build: (1) for each subformula  $\phi$  of  $\phi^*$ , starting from the innermost atomic propositions, an  $\mathbf{E}V^+\mathbf{MDD}$  encoding the size  $\pi_\phi(\mathbf{i})$  of a minimum witness for  $\phi$  starting from each state  $\mathbf{i}$ , and (2) for each subformula  $\mathbf{E}G\phi$  of  $\phi^*$ , an  $\mathbf{E}V^+\mathbf{MDD}2$  encoding the size,  $TC_\phi(\mathbf{i}, \mathbf{j})$ , of a minimum witness for a path of states satisfying  $\phi$  from each state  $\mathbf{i}$  to each state  $\mathbf{j}$ .

We present algorithms to compute the minimum witness size for  $\mathbf{E}U$  and  $\mathbf{E}G$ , while we omit the simpler ones for logical operators,  $\mathbf{E}X$ , and atomic propositions.

---

```

EUSat(EV+MDD  $\langle \alpha_\rho, p_\rho \rangle$ , EV+MDD  $\langle \beta_\phi, q_\phi \rangle$ )
1:  $\langle \mu, u \rangle \leftarrow \text{ConsSat}(\langle \alpha_\rho, p_\rho \rangle, \langle \beta_\phi, q_\phi \rangle)$ 
2: return  $\langle \mu, u \rangle$ 

```

---

```

ConsSat(EV+MDD  $\langle \alpha, p \rangle$ , EV+MDD  $\langle \beta, q \rangle$ )
1: if  $\alpha = \infty$  or  $\beta = \infty$  then return  $\langle \infty, \Omega \rangle$ 
2:  $k \leftarrow p.lvl$  ▷ we assume quasi-reduced rule, thus  $p.lvl = q.lvl$ 
3: if  $l = 0$  then return  $\langle \alpha, \Omega \rangle$ 
4: if ConsSatGet( $p, \langle \beta, q \rangle, \langle \gamma, u \rangle$ ) then return  $\langle \alpha + \gamma, u \rangle$  ▷ ConsSat( $\langle 0, p \rangle, \langle \beta, q \rangle$ ) =  $\langle \gamma, u \rangle$ 
5:  $u \leftarrow \text{EVMDDNode}(k)$ 
6: for each  $i \in \mathcal{S}_k$  do
7:   if  $q[i].v = \infty$  then  $u[i] \leftarrow p[i]$ 
8:   else  $u[i] \leftarrow \text{ConsSat}(p[i], \langle \beta + q[i].v, q[i].c \rangle)$ 
9: repeat
10:   for each  $i, j \in \mathcal{S}_k$  do
11:      $\langle \tau, t \rangle \leftarrow \text{ConsRelProdSat}(\langle \alpha + u[i].v, u[i].c \rangle, \langle \beta + q[j].v, q[j].c \rangle, \mathcal{N}_k^{-1}[i][j])$ 
12:      $u[j] \leftarrow \text{Min}(u[j], \langle \tau, t \rangle)$ 
13: until  $u$  does not change
14:  $\langle \mu, u \rangle \leftarrow \text{Normalize}(u)$ 
15: ConsSatPut( $p, \langle \beta, q \rangle, \langle \mu - \alpha, u \rangle$ ) ▷ memoize the result
16: return  $\langle \mu, u \rangle$ 

```

---

```

ConsRelProdSat(EV+MDD  $\langle \alpha, p \rangle$ , EV+MDD  $\langle \beta, q \rangle$ , MDD2  $r$ )
1: if  $\alpha = \infty$  or  $\beta = \infty$  or  $r = 0$  then return  $\langle \infty, \Omega \rangle$ 
2:  $k \leftarrow p.lvl$  ▷ we assume quasi-reduced rule, thus  $p.lvl = q.lvl$ 
3: if  $l = 0$  then return  $\langle \alpha + \beta, \Omega \rangle$ 
4: if ConsRelProdSatGet( $p, \langle \beta, q \rangle, r, \langle \gamma, u \rangle$ ) then return  $\langle \alpha + \gamma, u \rangle$ 
5:  $u \leftarrow \text{EVMDDNode}(k)$ 
6: for each  $i, j \in \mathcal{S}_k$  do
7:    $\langle \tau, t \rangle \leftarrow \text{ConsRelProdSat}(\langle \alpha + p[i].v, p[i].c \rangle, \langle \beta + q[j].v, q[j].c \rangle, r[i][j])$ 
8:    $u[j] \leftarrow \text{Min}(u[j], \langle \tau, t \rangle)$ 
9:  $\langle \mu, u \rangle \leftarrow \text{Normalize}(u)$ 
10:  $\langle \mu, u \rangle \leftarrow \text{ConsSat}(\langle \mu, u \rangle, \langle \beta, q \rangle)$ 
11: ConsRelProdSatPut( $p, \langle \beta, q \rangle, r, \langle \mu - \alpha, u \rangle$ ) ▷ memoize the result
12: return  $\langle \mu, u \rangle$ 

```

---

**Fig. 3.** Algorithm to compute the minimum witness size for **EU** formulas.

#### 4.1 Computing the Minimum Witness Size for **EU** Formulas

In [15], we introduced a “constrained” variant of saturation that restricts exploration to states satisfying a given property. Instead of applying “after-the-fact” intersections, this approach employs a “check-and-fire” policy, firing an event only when the next states to be obtained satisfy the given property, through an on-the-fly check. Now, we further extend this idea to take into account the sizes of subwitnesses demonstrating the satisfaction of subformulas.

*EUSat* in Fig. 3 is the top-level procedure to compute  $\pi_{\mathbf{E}\phi\mathbf{U}\rho}$ , given  $\langle\alpha_\rho, p_\rho\rangle$  encoding  $\pi_\rho$  and  $\langle\beta_\phi, q_\phi\rangle$  encoding  $\pi_\phi$  (both obtained by computing the minimum witness size function of subformulas). *ConsSat* computes a fixpoint for the subfunction encoded by  $\langle\alpha, p\rangle$ , under constraint  $\langle\beta, q\rangle$ , w.r.t. events affecting variables up to  $p$ 's level. *ConsRelProdSat* first recursively computes the  $\langle\beta, q\rangle$ -constrained relational product of  $\langle\alpha, p\rangle$  and  $r$  (specifically, it serves as a constrained version of the pre-image operation since we use the previous-state function), then it saturates the resulting node to ensure that it reaches a local fixpoint.

When exploring the pre-image of state  $\mathbf{i}$ , we compute, for each predecessor  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ , the sum of  $\pi_\rho(\mathbf{j})$  and the value currently associated to  $\mathbf{i}$  (line 3 in *ConsRelProdSat*), and use it to reduce the value associated to  $\mathbf{j}$ , if smaller (line 12 in *ConsSat* and line 8 in *ConsRelProdSat*). Upon reaching a fixpoint, we have the size of the minimum tree-like  $\mathbf{E}\phi\mathbf{U}\rho$  witness for each state  $\mathbf{i}$  ( $\infty$  if  $\mathbf{i} \not\models \mathbf{E}\phi\mathbf{U}\rho$ ).

The hash-key for the cache entries of *ConsSat* and *ConsRelProdSat* consists of two nodes,  $p$  and  $q$ , plus the value  $\beta$  attached to the edge for  $q$ , representing the constraint. Storing the *difference*  $\alpha - \beta$ , as done for *Min*, would be incorrect because saturation computes local fixpoints and thus may fire an event multiple times in one call, and  $\alpha$  just serves as an offset to all the values in the final function, while, for each individual state, it does not affect whether the new value obtained from one firing is smaller than the currently associated value. In other words, if it is known that  $\text{ConsSat}(\langle 0, p \rangle, \langle \beta, q \rangle) = \langle \gamma, u \rangle$ , then we can conclude that  $\text{ConsSat}(\langle \alpha, p \rangle, \langle \beta, q \rangle) = \langle \alpha + \gamma, u \rangle$ , for any  $\alpha > 0$ .

### 4.2 Computing the Minimum Witness Size for EG Formulas

A witness of  $\mathbf{E}Gp$  is a lasso-shaped infinite path consisting of a finite prefix leading to a cycle [1]. Thus, two steps are needed to compute a minimum tree-like witness for  $\mathbf{E}G\phi$ : (1) identify all states in cycles of states satisfying  $\phi$ , and their minimum witness size; (2) starting from these states, explore the model backward to find all states satisfying  $\mathbf{E}Gp$ , and their minimum witness size. This second step is essentially an  $\mathbf{E}U$  computation, thus we focus on the first step.

Given a graph, the transitive closure (TC) describes the reachability between any pair of nodes. Computing TCs was deemed infeasible for large models [13], but recent attempts using saturation to compute TCs symbolically have been successful [16, 17]. We generalize this approach so that, for  $\mathbf{E}G\phi$ , the size of a minimum  $\phi$  witness for each state in a  $\phi$  cycle contributes to the cycle size.

We compute function  $TC_\phi : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{N}_\infty$  s.t.  $TC_\phi(\mathbf{i}, \mathbf{j})$  is the minimum size of any path  $\mathbf{i}, \mathbf{i}_1, \dots, \mathbf{j}$ , computed as  $\pi_\phi(\mathbf{i}_1) + \dots + \pi_\phi(\mathbf{j})$ , or  $\infty$  if no such path exists. We do not include  $\pi_\phi(\mathbf{i})$  because we compute  $TC_\phi$  to obtain the minimum witness size of cycles,  $\chi_\phi(\mathbf{i}) = TC_\phi(\mathbf{i}, \mathbf{i}) + 1$ , and  $\pi_\phi(\mathbf{i})$  should not be added twice.

The procedure to compute  $TC_\phi$  is analogous to a symbolic implementation of Dijkstra's algorithm in a weighted graph. We initialize the function

$$\lambda_\phi(\mathbf{i}_1, \mathbf{i}_2) = \begin{cases} \pi_\phi(\mathbf{i}_2) & \text{if } \mathbf{i}_2 \in \mathcal{N}(\mathbf{i}_1), \mathbf{i}_1 \models \phi, \text{ and } \mathbf{i}_2 \models \phi \\ \infty & \text{otherwise} \end{cases}$$



and repeat the following computation until reaching a fixpoint:

$$\lambda_\phi(\mathbf{i}_1, \mathbf{i}_2) = \min\{\lambda_\phi(\mathbf{i}_1, \mathbf{i}_2), \min\{\lambda_\phi(\mathbf{i}_1, \mathbf{i}) + \pi_\phi(\mathbf{i}_2) : \forall \mathbf{i} \in \mathcal{N}^{-1}(\mathbf{i}_2)\}\}.$$

This iteration never increases the value of  $\lambda_\phi$ , thus it terminates, and when it does the resulting fixpoint is  $TC_\phi$ . Procedures  $TCSat$  and  $TCRelProdSat$  in Fig. 4 are similar to  $ConsSat$  and  $ConsRelProdSat$  in Fig. 3, except that they apply saturation to an EV+MDD2.  $TCMin$  (line 14 in  $TCSat$ , line 11 in  $TCRelProdSat$ ) is an implementation of  $Min$  over pairs of states: for each  $\mathbf{i}, \mathbf{j} \in \mathcal{S}$ ,  $TCMin_{f,g}(\mathbf{i}, \mathbf{j}) = \min\{f(\mathbf{i}, \mathbf{j}), g(\mathbf{i}, \mathbf{j})\}$ .

Finally, we compute  $\pi_{EG\phi}$  with procedure  $EGSat$ , where  $\langle \beta_\phi, q_\phi \rangle$  encodes  $\pi_\phi$ .  $BuildLambda$  (line 1) builds an EV+MDD2 encoding function  $\lambda_\phi$ , to initialize the computation of  $TC_\phi$ .  $ExtractCycles$  (line 2) returns an EV+MDD encoding  $\chi_\phi(\mathbf{i})$  by extracting elements  $TC_\phi(\mathbf{i}, \mathbf{i})$  from  $TC_\phi$ , for  $\mathbf{i} \in \mathcal{S}$ , and adding 1 to them.

## 5 Generating a Minimum Tree-Like Witness

Recall that, if function  $f$  is encoded as an EV+MDD, one can retrieve  $MinVal(f)$ , the minimum value of  $f$ , in constant time (as the value labeling the edge pointing to the root node) and  $MinState(f)$ , a state achieving that minimum value, in time proportional to the number of levels  $L$  (follow a path of 0-valued edges from the root to  $\Omega$ ). Evaluating  $f(\mathbf{i})$  for a given state  $\mathbf{i}$  also requires just  $L$  steps.

To obtain a minimum overall witness recursively, we start from an initial state  $\mathbf{i}^*$  with a minimum witness for  $\phi^*$ . This state can be found by building an EV+MDD encoding the function  $f_{init} : \mathcal{S} \rightarrow \{0, \infty\}$  evaluating to 0 iff  $\mathbf{i} \in \mathcal{S}_{init}$ , and then the EV+MDD encoding  $\pi_{\phi^* \wedge init}$ , the elementwise maximum of  $f_{init}$  and  $\pi_{\phi^*}$  (i.e., the restriction of the minimum witness size for  $\phi^*$  to the initial states). If  $MinVal(\pi_{\phi^* \wedge init}) = \infty$ , no initial state satisfies  $\phi^*$ , thus no witness exists. Otherwise, there is a minimum witness of size  $MinVal(\pi_{\phi^* \wedge init})$  starting from an initial state  $\mathbf{i}^* = MinState(\pi_{\phi^* \wedge init})$ , and the call  $MinWit(\mathbf{i}^*, \phi^*, \pi_{\phi^*}(\mathbf{i}^*))$  (see Fig. 5) will recursively generate one such minimum witness.

## 6 Experimental Results

We implemented both our EV+MDD-based approach to generate minimum tree-like witnesses (MINWIT) and the traditional MDD-aided BFS approach [7] to generate (not necessarily minimum) tree-like witnesses (WIT) in our model checker SMART [2]. Then we ran them on a benchmark suite consisting of nine Petri net models from the 2017 Model Checking Contest (<https://mcc.lip6.fr/2017/>). All models have one or more scaling parameters affecting the number of states and state-to-state transitions, thus the model size and complexity. To generate tree-like witnesses, we define an ECTL formula that the model satisfies (the specific formula is listed in the results presented in Table 1). The datasets we utilized are available in the figshare repository [9].

---

```

EGSat( $EV^+MDD \langle \beta_\phi, q_\phi \rangle$ )
1:  $\langle \tau, t \rangle \leftarrow TCSat(BuildLambda(\langle \beta_\phi, q_\phi \rangle), \langle \beta_\phi, q_\phi \rangle)$ 
2:  $\langle \mu, u \rangle \leftarrow ConsSat(ExtractCycles(\langle \tau, t \rangle), \langle \beta_\phi, q_\phi \rangle)$ 
3: return  $\langle \mu, u \rangle$ 

```

---

```

TCSat( $EV^+MDD2 \langle \alpha, p \rangle, EV^+MDD \langle \beta, q \rangle$ )
1: if  $\alpha = \infty$  or  $\beta = \infty$  then return  $\langle \infty, \Omega \rangle$ 
2:  $k \leftarrow p.lvl$   $\triangleright$  we assume quasi-reduced rule, thus  $p.lvl = q.lvl$ 
3: if  $k = 0$  then return  $\langle \alpha, \Omega \rangle$ 
4: if TCSatGet( $p, \langle \beta, q \rangle, \langle \gamma, u \rangle$ ) then return  $\langle \alpha + \gamma, u \rangle$   $\triangleright TCSat(\langle 0, p \rangle, \langle \beta, q \rangle) = \langle \gamma, u \rangle$ 
5:  $u \leftarrow EVMDD2Node(k)$ 
6: for each  $i, j \in S_k$  do
7:    $u[i].c[j] \leftarrow TCSat(\langle \alpha + p[i].v + p[i].c[j].v, p[i].c[j].c \rangle, \langle \beta + q[j].v, q[j].c \rangle)$ 
8: for each  $i \in S_k$  do
9:    $w \leftarrow EVMDD2Node(Prime(k))$ 
10:  repeat
11:    for each  $j, j' \in S_k$  do
12:       $\langle \alpha', u' \rangle \leftarrow \langle \alpha + u[i].v + u[i].c[j].v, u[i].c[j].c \rangle$ 
13:       $\langle \tau, t \rangle \leftarrow TCRelProdSat(\langle \alpha', u' \rangle, \langle \beta + q[j'].v, q[j'].c \rangle, \mathcal{N}_k[j][j'])$ 
14:       $w[j'] \leftarrow TCMIn(w[j'], \langle \tau, t \rangle)$ 
15:    until  $w$  does not change
16:     $u[i] \leftarrow Normalize(w)$ 
17:  $\langle \mu, u \rangle \leftarrow Normalize(u)$ 
18: TCSatPut( $p, \langle \beta, q \rangle, \langle \mu - \alpha, u \rangle$ )  $\triangleright$  memoize the result
19: return  $\langle \mu, u \rangle$ 

```

---

```

TCRelProdSat( $EV^+MDD2 \langle \alpha, p \rangle, EV^+MDD \langle \beta, q \rangle, MDD2 r$ )
1: if  $\alpha = \infty$  or  $\beta = \infty$  or  $r = 0$  then return  $\langle \infty, \Omega \rangle$ 
2:  $k \leftarrow p.lvl$   $\triangleright$  we assume quasi-reduced rule, thus  $p.lvl = q.lvl$ 
3: if  $k = 0$  then return  $\langle \alpha + \beta, \Omega \rangle$ 
4: if TCRelProdSatGet( $p, \langle \beta, q \rangle, r, \langle \gamma, u \rangle$ ) then return  $\langle \alpha + \gamma, u \rangle$ 
5:  $u \leftarrow EVMDD2Node(k)$ 
6: for each  $i \in S_k$  do
7:    $w \leftarrow EVMDD2Node(Prime(k))$ 
8:   for each  $j, j' \in S_k$  do
9:      $\langle \alpha', p' \rangle \leftarrow \langle \alpha + p[i].v + p[i].c[j].v, p[i].c[j].c \rangle$ 
10:     $\langle \tau, t \rangle \leftarrow TCRelProdSat(\langle \alpha', p' \rangle, \langle \beta + q[j'].v, q[j'].c \rangle, r[j][j'])$ 
11:     $w[j'] \leftarrow TCMIn(w[j'], \langle \tau, t \rangle)$ 
12:    $u[i] \leftarrow Normalize(w)$ 
13:  $\langle \mu, u \rangle \leftarrow Normalize(u)$ 
14:  $\langle \mu, u \rangle \leftarrow TCSat(\langle \mu, u \rangle, \langle \beta, q \rangle)$ 
15: TCRelProdSatPut( $p, \langle \beta, q \rangle, r, \langle \mu - \alpha, u \rangle$ )  $\triangleright$  memoize the result
16: return  $\langle \mu, u \rangle$ 

```

---

**Fig. 4.** Algorithm to compute the minimum witness size for **EG** formulas.

---

```

MinWit(state i, ECTLformula  $\phi$ , size  $n$ )
1: if  $\phi \in \mathcal{A}$  then return i ▷  $n = 1$ ,  $\phi$  is an atomic proposition
2: if  $\phi = \phi' \wedge \rho'$  then ▷  $n = \pi_{\phi'}(\mathbf{i}) + \pi_{\phi'}(\mathbf{j}) - 1$ 
3:   return  $\llbracket \text{MinWit}(\mathbf{i}, \phi', \pi_{\phi'}(\mathbf{i})) \rrbracket \diamond \llbracket \text{MinWit}(\mathbf{i}, \rho', \pi_{\rho'}(\mathbf{i})) \rrbracket$ 
4: if  $\phi = \phi' \vee \rho'$  then ▷  $n = \min\{\pi_{\phi'}(\mathbf{i}), \pi_{\rho'}(\mathbf{i})\}$ 
5:   if  $n = \pi_{\phi'}(\mathbf{i})$  then return  $\llbracket \text{MinWit}(\mathbf{i}, \phi', \pi_{\phi'}(\mathbf{i})) \rrbracket$ 
6:   else return  $\llbracket \text{MinWit}(\mathbf{i}, \rho', \pi_{\rho'}(\mathbf{i})) \rrbracket$ 
7: if  $\phi = \text{EX}\phi'$  then
8:   choose  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$  s.t.  $\pi_{\phi'}(\mathbf{j}) = n - 1$  ▷ there exists at least one such  $\mathbf{j}$ 
9:   return  $\mathbf{i}, \llbracket \text{MinWit}(\mathbf{j}, \phi', n - 1) \rrbracket$ 
10: if  $\phi = \text{E}\phi' \cup \rho'$  then
11:   if  $\pi_{\rho'}(\mathbf{i}) = n$  then ▷ a minimum witness for  $\mathbf{i} \models \rho'$  works for  $\mathbf{i} \models \text{E}\phi' \cup \rho'$ 
12:     return  $\llbracket \text{MinWit}(\mathbf{i}, \rho', n) \rrbracket$ 
13:   else ▷ no witness for  $\mathbf{i} \models \rho'$  is minimum for  $\mathbf{i} \models \text{E}\phi' \cup \rho'$ 
14:     choose  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$  s.t.  $\pi_{\phi'}(\mathbf{j}) = n - \pi_{\phi'}(\mathbf{i})$  ▷ there exists at least one such  $\mathbf{j}$ 
15:     return  $\llbracket \text{MinWit}(\mathbf{i}, \phi', \pi_{\phi'}(\mathbf{i})) \rrbracket, \llbracket \text{MinWit}(\mathbf{j}, \phi, n - \pi_{\phi'}(\mathbf{i})) \rrbracket$ 
16: if  $\phi = \text{EG}\phi'$  then
17:   if  $TC_{\phi'}(\mathbf{i}, \mathbf{i}) = n - 1$  then ▷ a minimum cycle witness for  $\mathbf{i}$  works for  $\mathbf{i} \models \text{EG}\phi'$ 
18:     return  $\text{CloseCycle}(\mathbf{i}, \mathbf{i}, \phi', n - 1)$ 
19:   else ▷  $\mathbf{i}$  is on the handle of a lasso for a minimum witness for  $\mathbf{i} \models \text{EG}\phi'$ 
20:     choose  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$  s.t.  $\pi_{\phi'}(\mathbf{j}) = n - \pi_{\phi'}(\mathbf{i})$  ▷ there exists at least one such  $\mathbf{j}$ 
21:     return  $\llbracket \text{MinWit}(\mathbf{i}, \phi', \pi_{\phi'}(\mathbf{i})) \rrbracket, \llbracket \text{MinWit}(\mathbf{j}, \phi, n - \pi_{\phi'}(\mathbf{i})) \rrbracket$ 

```

---

```

CloseCycle(state  $\mathbf{j}$ , state  $\mathbf{i}$ , ECTLformula  $\phi$ , size  $n$ )
1: if  $\pi_{\phi}(\mathbf{i}) = n$  then ▷ it must be that  $\mathbf{i} \in \mathcal{N}(\mathbf{j})$ , close the cycle with  $\bar{\mathbf{i}}$ 
2:   return  $\llbracket \text{MinWit}(\mathbf{j}, \phi, \pi_{\phi}(\mathbf{j})) \rrbracket, \bar{\mathbf{i}}$ 
3: else
4:   choose  $\mathbf{k} \in \mathcal{N}(\mathbf{j})$  s.t.  $TC_{\phi}(\mathbf{k}, \mathbf{i}) = n - \pi_{\phi}(\mathbf{k})$  ▷ there exists at least one such  $\mathbf{k}$ 
5:   return  $\llbracket \text{MinWit}(\mathbf{j}, \phi, \pi_{\phi}(\mathbf{j})) \rrbracket, \text{CloseCycle}(\mathbf{k}, \mathbf{i}, \phi, n - \pi_{\phi}(\mathbf{k}))$ 

```

---

**Fig. 5.** Algorithm to generate a minimum tree-like witness.

For MINWIT, we run each model instance with a timeout of one hour, and report the runtime, the peak memory consumption, and the size of the minimum witness. For WIT, we run each instance 100 times and report the total runtime, the peak memory consumption, and the minimum, average and maximum size among the all the generated witnesses. The minimum witness size is in bold when WIT did not manage to generate a minimum witness in any of the 100 runs. Obviously, the choice of  $R = 100$  runs is arbitrary: the larger  $R$  is, the more likely WIT is to generate smaller witnesses, possibly a minimum one, but, on the other hand, the overall time WIT spends for witness generation is roughly proportional to  $R$ . Fundamentally, however, we have no easy way to know if the smallest witness generated by WIT is a minimum one, regardless of how large  $R$  is, while MINWIT guarantees minimality.

A few observations are in order. First, it is not surprising that MINWIT is sometimes orders of magnitude slower and requires more memory than WIT.

**Table 1.** Performance comparison of MINWIT and WIT.

| Model (parms)  | #States             | #Trans              | Time (s) |         | Memory (MB) |        | Size        |      |      |      |
|--|---------------------|---------------------|----------|---------|-------------|--------|-------------|------|------|------|
|  |                     |                     | MINWIT   | WIT     | MINWIT      | WIT    | MINWIT      | WIT  |      |      |
|  |                     |                     |          |         |             |        |             | min  | avg  | max  |
| <b>EG(EF((Section_2 = 1) ∧ (Section_3 = 1)))</b>                                 |                     |                     |          |         |             |        |             |      |      |      |
| CircularTrain(12)  | $2.0 \cdot 10^2$    | $5.0 \cdot 10^2$    | 0.4      | 0.0     | 20.6        | 4.8    | <b>25</b>   | 32   | 71   | 275  |
| CircularTrain(24)  | $8.7 \cdot 10^4$    | $4.1 \cdot 10^5$    | 2244.9   | 4.6     | 2924.5      | 11.8   | <b>37</b>   | 91   | 404  | 889  |
| <b>E(EF(ERKPP &gt; 5) U EG(RKIPP_RP &gt; 5))</b>                                 |                     |                     |          |         |             |        |             |      |      |      |
| ERK(20)  | $1.7 \cdot 10^6$    | $1.6 \cdot 10^7$    | 93.9     | 3.8     | 591.0       | 6.5    | <b>129</b>  | 258  | 313  | 391  |
| ERK(22)  | $2.8 \cdot 10^6$    | $2.7 \cdot 10^7$    | 224.1    | 4.4     | 932.1       | 6.9    | <b>129</b>  | 246  | 314  | 393  |
| ERK(25)  | $5.7 \cdot 10^6$    | $5.4 \cdot 10^7$    | 793.4    | 5.0     | 1800.6      | 8.0    | <b>129</b>  | 256  | 315  | 397  |
| <b>EF((P1 = 3) ∧ EG((P1 &gt; P2) ∧ (P2 &gt; P3)))</b>                            |                     |                     |          |         |             |        |             |      |      |      |
| FMS(5)   | $2.9 \cdot 10^6$    | $3.2 \cdot 10^7$    | 0.6      | 2.7     | 21.0        | 7.6    | 13          | 13   | 48   | 193  |
| FMS(8)   | $2.5 \cdot 10^8$    | $3.6 \cdot 10^9$    | 31.6     | 17.5    | 447.6       | 13.7   | 22          | 22   | 51   | 201  |
| FMS(10)  | $2.5 \cdot 10^9$    | $4.1 \cdot 10^{10}$ | 458.9    | 30.6    | 1510.1      | 23.5   | 28          | 28   | 54   | 217  |
| <b>EF((P1 &lt; P2) ∧ EG(P1 = P4))</b>  |                     |                     |          |         |             |        |             |      |      |      |
| Kanban(20)   | $8.1 \cdot 10^{11}$ | $1.1 \cdot 10^{13}$ | 18.4     | 1530.3  | 269.6       | 289.0  | 10          | 10   | 10   | 11   |
| Kanban(22)   | $2.1 \cdot 10^{12}$ | $2.9 \cdot 10^{13}$ | 28.6     | 2297.8  | 395.9       | 410.4  | 10          | 10   | 10   | 11   |
| Kanban(25)   | $7.7 \cdot 10^{12}$ | $1.1 \cdot 10^{14}$ | 53.9     | 4224.4  | 691.6       | 707.9  | 10          | 10   | 10   | 11   |
| <b>E(EF(Phase1 &lt; Phase2) U (Phase2 &gt; Phase3))</b>                          |                     |                     |          |         |             |        |             |      |      |      |
| MAPK(8)  | $6.1 \cdot 10^6$    | $7.9 \cdot 10^7$    | 14.0     | 2.0     | 353.0       | 6.1    | <b>70</b>   | 126  | 126  | 126  |
| MAPK(12)   | $3.2 \cdot 10^8$    | $5.0 \cdot 10^9$    | 1881.4   | 6.2     | 1764.2      | 8.7    | <b>109</b>  | 204  | 204  | 204  |
| <b>EF((Think_1 = 0) ∧ EG(Eat_1 = 0))</b>   |                     |                     |          |         |             |        |             |      |      |      |
| Philosophers(20)   | $3.5 \cdot 10^9$    | $5.4 \cdot 10^{10}$ | 1.3      | 2.1     | 52.4        | 9.2    | 5           | 5    | 8    | 22   |
| Philosophers(50)   | $7.2 \cdot 10^{23}$ | $2.8 \cdot 10^{25}$ | 44.9     | 10.8    | 763.5       | 29.1   | 5           | 5    | 7    | 20   |
| Philosophers(100)  | $5.2 \cdot 10^{47}$ | $4.0 \cdot 10^{49}$ | timeout  | 52.1    | –           | 94.0   | –           | 5    | 7    | 28   |
| <b>E(EF(P_client_ack_1 = 1) U ((P_server_ack_1 = 1) ∧ (P_server_ack_2 = 1)))</b> |                     |                     |          |         |             |        |             |      |      |      |
| SimpleLoadBal(2)   | $8.3 \cdot 10^2$    | $3.4 \cdot 10^3$    | 0.1      | 0.8     | 9.0         | 5.9    | 23          | 23   | 32   | 44   |
| SimpleLoadBal(5)   | $1.2 \cdot 10^5$    | $7.5 \cdot 10^5$    | 37.6     | 19.2    | 1032.6      | 41.0   | <b>23</b>   | 26   | 69   | 80   |
| <b>E(EF(TaskOnDisk &lt; CPUUnit) U (CPUUnit &lt; DiskControllerUnit))</b>        |                     |                     |          |         |             |        |             |      |      |      |
| SmallOS(64,32)   | $9.1 \cdot 10^6$    | $6.8 \cdot 10^7$    | 17.5     | 1987.7  | 374.6       | 401.6  | <b>662</b>  | 694  | 1189 | 1552 |
| SmallOS(128,64)  | $2.6 \cdot 10^8$    | $2.0 \cdot 10^9$    | 294.4    | 53522.2 | 3228.4      | 1850.0 | <b>2342</b> | 2430 | 4698 | 5920 |
| <b>EF(EG(Undress &lt; InBath))</b>   |                     |                     |          |         |             |        |             |      |      |      |
| SwimmingPool(1)  | $9.0 \cdot 10^4$    | $4.5 \cdot 10^5$    | 109.7    | 4.7     | 1334.9      | 6.6    | 16          | 16   | 24   | 43   |
| SwimmingPool(2)  | $3.4 \cdot 10^6$    | $2.0 \cdot 10^7$    | timeout  | 39.1    | –           | 22.3   | –           | 16   | 24   | 53   |

Building EV<sup>+</sup>MDDs or EV<sup>+</sup>MDD2s encoding both states and size information is much more expensive than the image computations on MDDs used to just run the model checking phase, as WIT does. However, this is offset by a minimality guarantee. Interestingly, there are cases where MINWIT completes with a runtime and memory consumption comparable to a single run of WIT (e.g., Kanban) or even faster (e.g., SmallOS). We give credit to the saturation algorithm for its efficient locality-exploiting exploration.

Second, for models where small, simple witnesses exist, WIT may be able to generate a minimum witness. Since the backward exploration guarantees the

local minimality of subwitnesses for **EX**, **EF** and **EU** segments, such greedy strategy may result in a global minimum witness, determined by the structure of the model. But this occurrence cannot be guaranteed, regardless of whether we use 100 runs or 10,000 runs, so that, even when WIT happens to generate a minimum witness, users do not know that this is indeed the case.

Third, for models where only large, complex witnesses exist, generating a minimum witness is almost impossible for WIT, while the witness from MINWIT can be only 40% as large as the smallest one generated by WIT (e.g., Circular-Train with  $N = 24$ ). Additionally, WIT's greedy strategy may trap itself into a local optimum. For example, the ECTL formula used for model MAPK does not contain **EG**, and the minimum, average, and maximum witness sizes generated by WIT are equal, implying that WIT is unaware of other possibilities when it chooses branching states. Adopting a probabilistic non-optimal strategy like simulated annealing may alleviate this problem, but it still would not provide guarantees and would likely require many more runs.

The main limitation of MINWIT is that, since computing the minimum witness size function is computationally intensive, long runtimes and large amounts of memory are required as the model complexity scales up. However, engineers usually debug models with small scaling parameters first, perhaps running model checking tools overnight, thus, the resource requirement of MINWIT may often be acceptable in practice. In real-world applications, we believe that MINWIT and WIT can complement each other. WIT generates a large number of witnesses in a short time, but if all the witnesses are complex, MINWIT can be run to find a smaller, easier-to-inspect one. Conversely, if MINWIT fails to generate a minimum witness due to time or memory limitation, WIT can be run to obtain not-necessarily-minimum ones by running it repeatedly, as many times as one can afford. The best approach, given enough resources and in the presence of critical deadlines, may well be to run both methods in parallel, so that we can be sure to have a minimum witness if MINWIT completes, but we have at least some witnesses from WIT, if MINWIT fails to complete in time.

## 7 A Comment About Our Definition of Witness Size

In Sect. 3, we defined the witness size as “the number of appearances of states in the unfolded tree-like witness”. An alternative definition could have been “the number of distinct states in the unfolded tree-like witness”. However, a state may appear multiple times for different purposes in a witness. For example, the witness for  $\mathbf{EF}(a \wedge \mathbf{EG}b)$  in Fig. 6 contains state 2 twice, one in  $\llbracket 3, 2, \bar{3} \rrbracket$  to verify for the **EG** fragment, the other in  $\llbracket 1, 2, 3 \rrbracket$  to verify the **EF** fragment. Considering each appearance separately makes each subpath independently verifiable, while merging states that appear multiple times and counting only distinct states loses this information. Admittedly, our definition of witness size also enables our approach to obtain minimum witnesses, while we do not know a practical algorithm that can derive minimum witnesses according to a definition of size where common paths and states appearing multiple times are counted without repetition.

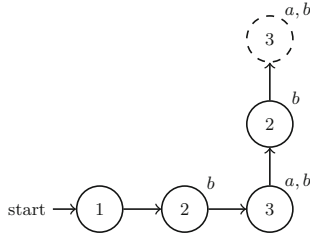


Fig. 6. A witness for  $\mathbf{EF}(a \wedge \mathbf{EG}b)$ .

Of course, after obtaining a minimum witness according to our definition, we could attempt to merge shared portions among subpaths that demonstrate the same property, but doing so would not generally result in minimum witnesses according to this alternative view of witness size.

### 8 Conclusions

We presented a definition of witness size and an approach to compute minimum tree-like witnesses for ECTL formulas, based on edge-valued decision diagrams to capture a global view of witness size. Experimental results demonstrate that our approach is able to generate minimum witnesses (with a guarantee that it is doing so) for some models, while the traditional approach is not. While the runtime and memory requirements of our approach tend to be higher, sometimes they are comparable to that of the traditional approach.

There are several directions for future work to improve this approach itself or extend its applicability. One interesting possibility is to selectively employ our approach or the traditional approach for different subformulas; this would not guarantee witness minimality, but could generate smaller witness than with the traditional approach alone, while being faster than using our approach alone. Especially for **EG** formulas, the traditional approach has no global view about the size of witnesses it generates, while, for formulas where the minimum witness size from each state varies widely, the  $\mathbf{EV}^+\mathbf{MDD}$ s and  $\mathbf{EV}^+\mathbf{MDD}2$ s built by our method tend to be large and costly to compute. Thus, heuristics that consider both the structure of the model and of the formula are needed.

Finally, our approach could be further extended by generalizing the concept of “size” to “weight”. Specifically, by assigning a weight to each state, engineers could convey their preference to model checkers, which would then tend to generate witnesses containing the desired states and subpaths, instead of just counting the number of states in the witness. The algorithmic difference in doing so would be negligible, the only additional cost could be a potential growth in the size of the corresponding  $\mathbf{EV}^+\mathbf{MDD}$ s and  $\mathbf{EV}^+\mathbf{MDD}2$ s, as the functions being encoded might have less sharing of nodes.

**Data Availability Statement and Acknowledgments.** The datasets we utilized are available in the figshare repository [9]. This work was supported in part by the National Science Foundation under grant ACI-1642397.

## References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-49059-0\\_14](https://doi.org/10.1007/3-540-49059-0_14)
2. Ciardo, G., Jones, R.L., Miner, A.S., Siminiceanu, R.: Logical and stochastic modeling with SMART. *Perf. Eval.* **63**, 578–608 (2006)
3. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer* **8**(1), 4–25 (2006)
4. Ciardo, G., Siminiceanu, R.: Using edge-valued decision diagrams for symbolic generation of shortest paths. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 256–273. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-36126-X\\_16](https://doi.org/10.1007/3-540-36126-X_16)
5. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 146–161. Springer, Heidelberg (2005). [https://doi.org/10.1007/11560548\\_13](https://doi.org/10.1007/11560548_13)
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) *Logic of Programs* 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). <https://doi.org/10.1007/BFb0025774>
7. Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: *Proceedings of the LICS*, pp. 19–29. IEEE Computer Society Press (2002)
8. Clarke, E.M., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: *32nd Design Automation Conference (DAC 1995)*, pp. 427–432 (1995)
9. Jiang, C., Ciardo, G.: Generation of minimum tree-like witnesses for existential CTL. *Figshare* (2018). <https://doi.org/10.6084/m9.figshare.5926555.v1>
10. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.: Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic* **4**(1–2), 9–62 (1998)
11. Kashyap, S., Garg, V.K.: Producing short counterexamples using “crucial events”. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 491–503. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-70545-1\\_47](https://doi.org/10.1007/978-3-540-70545-1_47)
12. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, May 1992. CMU-CS-92-131
13. Ravi, K., Bloem, R., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 162–179. Springer, Heidelberg (2000). [https://doi.org/10.1007/3-540-40922-X\\_10](https://doi.org/10.1007/3-540-40922-X_10)
14. Tan, J., Avrunin, G.S., Clarke, L.A., Zilberstein, S., Leue, S.: Heuristic-guided counterexample search in FLAVERS. In: *Proceedings of the SIGSOFT*, pp. 201–210. ACM (2004)

15. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 368–381. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-04761-9\\_27](https://doi.org/10.1007/978-3-642-04761-9_27)
16. Zhao, Y., Ciardo, G.: Symbolic computation of strongly connected components using saturation. In: Proceedings of the 2nd NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp. 201–211. NASA (2010)
17. Zhao, Y., Jin, X., Ciardo, G.: A symbolic algorithm for shortest EG witness generation. In: Proceedings of the TASE, pp. 68–75. IEEE Computer Society Press (2011)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

