Simplification of CTL Formulae for Efficient Model Checking of Petri Nets

Frederik Bønneland, Jakob Dyhr, Peter G. Jensen,
Mads Johannsen, and Jiří Srba

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark

Abstract. We study techniques to overcome the state space explosion problem in CTL model checking of Petri nets. Classical state space pruning approaches like partial order reductions and structural reductions become less efficient with the growing size of the CTL formula. The reason is that the more places and transitions are used as atomic propositions in a given formula, the more of the behaviour (interleaving) becomes relevant for the validity of the formula. We suggest several methods to reduce the size of CTL formulae, while preserving their validity. By these methods, we significantly increase the benefits of structural and partial order reductions, as the combination of our techniques can achive up to 60 percent average reduction in formulae sizes. The algorithms are implemented in the open-source verification tool TAPAAL and we document the efficiency of our approach on a large benchmark of Petri net models and queries from the Model Checking Contest 2017.

1 Introduction

Model checking [6] of distributed systems, described in high-level formalisms like Petri nets, is often a time and resource consuming task—attributed mainly to the state space explosion problem. Several techniques like partial order and symmetry reductions [16,20,21,23,24] and structural reductions [14,18,17] were suggested for reducing the size of the state space of a given Petri net in need of exploration to verify different logical specifications. These techniques try to prune the searchable state space and their efficiency is to a high degree influenced by the type and size of the logical formula in question. The larger the formula is and the more atomic propositions (querying the number of tokens in places or the fireability of certain transitions) it has, the less can be pruned away when exploring the state space and hence the effect of these techniques is reduced. It is therefore desirable to design techniques that can reduce the size of a given logical formula, while preserving the model checking answer. For practical applicability, it is important that such formula reduction techniques are computationally less demanding than the actual state space search.

In this paper, we focus on the well-known logic CTL [5] and describe three methods for CTL formula simplification, each preserving the logical equivalence

w.r.t. a the given Petri net model. The first two methods rely on standard logical equivalences of formulae, while the third one uses state equations of Petri nets and linear programming to recursively traverse the structure of a given CTL formula. During this process, we identify subformulae that are either trivially satisfied or impossible to satisfy, and we replace them with easier to verify alternatives. We provide an algorithm for performing such a formula simplification, including the traversal though temporal CTL operators, and prove the correctness of our approach.

The formula simplification methods are implemented and fully integrated into an open-source model checker TAPAAL [10] and its untimed verification engine `verifypn` [14]. We document the performance of our tool on the large benchmark of Petri net models and CTL queries from the Model Checking Contest 2017 (MCC'17) [15]. The data show that for CTL cardinality queries, we are able to achieve on average 60% of reduction of the query size and about 34% of queries are simplified into trivial queries *true* or *false*, hence avoiding completely the state space exploration. For CTL fireability queries, we achieved 50% reduction of the query size and about 10% of queries are simplified into *true* or *false*. Finally, we compare our simplification algorithm with the one implemented in the tool LoLA [26], the winner of MCC'17 in the several categories including the CTL category, documenting a noticeable performance margin in favour of our approach, both in the number of solved queries purely by the CTL simplification as well as when CTL verification follows the simplification process. For completeness, we also present the data for pure reachability queries where the tool Sara [25] (run parallel with LoLA during MCC'17) performs counterexample guided abstraction refinement and contributes to a high number (about twice as high as our tool) of solved reachability queries without the need to run LoLA's state space exploration. Nevertheless, if we also include the actual verification after the formula simplification, TAPAAL now moves 0.4% ahead of the combined performance of LoLA and Sara.

Related work. Traditionally, the conditions generated by the state equation technique [17] express linear constraints on the number of times the events can occur relative to other events of the system, and form a necessary condition for marking reachability. State equations were used in [14] as an over-approximation technique for preprocessing of reachability formulae in earlier editions of the model checking contest. As the technique can be often inconclusive, extensions of state equations were studied e.g. in [11] where the authors use traps to increase precision of the method, or in [8] where the state equation technique is extended to liveness properties. State equations, as a necessary condition for reachability, were also used in other application domains like concurrent programming [1,2]. Our work further extends state equations to full CTL logic and improves the precision of the method by a recursive evaluation of integer linear programs for all subformulae, while employing state equations for each subformula and its negation. State equations were also exploited in [22] in order to guide the state space search based on a minimal solution to the equations. This approach is orthogonal with ours as it essentially defines a heuristic search strategy that in the

worst case must explore the whole state space. More recently, the state equation technique was also applied to the coverability problem for Petri nets [4,12].

Formula rewriting techniques (in order to reduce the size of CTL formulae) are implemented in the tool LoLA [26]. The tool performs formula simplification by employing subformula rewriting rules that include a subset of the rules described in Section 3. LoLA also employs the model checking tool Sara [25] that uses state equations in combination with Counter Example Abstraction Refinement (CEGAR) to perform an exact reachability analysis, being able to answer both reachability and non-reachability questions and hence it is close to being a complete model checker. Sara shows a very convincing performance on reachability queries, however, in the CTL category, we are able to simplify to *true* or *false* almost twice as many formulae, compared to the combined performance of Sara and LoLA.

2 Preliminaries

A *labelled transition system* (LTS) is a tuple $TS = (\mathcal{S}, A, \rightarrow)$ where $\mathcal{S}$ is a set of states, A is a set of actions (or labels), and $\rightarrow \subseteq \mathcal{S} \times A \times \mathcal{S}$ is a transition relation. We write $s \xrightarrow{a} s'$ whenever $(s, a, s') \in \rightarrow$ and say that a is *enabled* in s. The set of all enabled actions in a state s is denoted $en(s)$. A state s is a *deadlock* if $en(s) = \emptyset$. We write $s \rightarrow s'$ whenever there is an action a such that $s \xrightarrow{a} s'$.

A *run* starting at s_0 is any finite or infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ where $s_0, s_1, s_2, \ldots \in \mathcal{S}$, $a_0, a_1, a_2 \cdots \in A$ and $(s_i, a_i, s_{i+1}) \in \rightarrow$ for all respective i. We use $\Pi(s)$ to denote the set of all runs starting at the state s. A run is *maximal* if it is either infinite or ends in a state that is a deadlock. Let $\Pi^{max}(s)$ denote the set of all maximal runs starting at the state s. A *position* i in a run $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ refers to the state s_i in the path and is written as π_i. If π is infinite then any i, $0 \leq i$, is a position in π. Otherwise $0 \leq i \leq n$ where s_n is the last state in π.

We now define the syntax and semantics of a *computation tree logic* (CTL) [7] as used in the Model Checking Contest [15]. Let AP be a set of atomic propositions. We evaluate atomic propositions on a given LTS $TS = (\mathcal{S}, A, \rightarrow)$ by the function $v : \mathcal{S} \rightarrow 2^{AP}$ so that $v(s)$ is the set of atomic propositions satisfied in the state $s \in \mathcal{S}$.

The CTL syntax is given as follows (where $\alpha \in AP$ ranges over atomic propositions):

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid \alpha \mid \mathit{deadlock} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A(\varphi_1 U \varphi_2) \mid E(\varphi_1 U \varphi_2) \ .$$

We use Φ_{CTL} to denote the set of all CTL formulae. The semantics of a CTL formula φ in a state $s \in \mathcal{S}$ is given in Table 1. We do not use only the minimal set of CTL operators because the query simplification tries to push the negation as far as possible to the atomic predicates. This significantly improves the performance of our on-the-fly CTL model checking algorithm and allows for a more refined query rewriting.

$s \models \mathit{true}$	
$s \not\models \mathit{false}$	
$s \models \alpha$	iff $\alpha \in v(s)$
$s \models \mathit{deadlock}$	iff $en(s) = \emptyset$
$s \models \varphi_1 \wedge \varphi_2$	iff $s \models \varphi_1$ and $s \models \varphi_2$
$s \models \varphi_1 \vee \varphi_2$	iff $s \models \varphi_1$ or $s \models \varphi_2$
$s \models \neg\varphi$	iff $s \not\models \varphi$
$s \models AX\varphi$	iff $s' \models \varphi$ for all $s' \in \mathcal{S}$ s.t. $s \to s'$
$s \models EX\varphi$	iff there is $s' \in \mathcal{S}$ s.t $s \to s'$ and $s' \models \varphi$
$s \models AF\varphi$	iff for all $\pi \in \Pi^{max}(s)$ there is a position i in π s.t. $\pi_i \models \varphi$
$s \models EF\varphi$	iff there is $\pi \in \Pi^{max}(s)$ and a position i in π s.t. $\pi_i \models \varphi$
$s \models AG\varphi$	iff for all $\pi \in \Pi^{max}(s)$ and for all positions i in π we have $\pi_i \models \varphi$
$s \models EG\varphi$	iff there is $\pi \in \Pi^{max}(s)$ s.t. for all positions i in π we have $\pi_i \models \varphi$
$s \models A(\varphi_1 U \varphi_2)$	iff for all $\pi \in \Pi^{max}(s)$ there is a position i in π s.t. $\pi_i \models \varphi_2$ and for all j, $0 \leq j < i$, we have $\pi_j \models \varphi_1$
$s \models E(\varphi_1 U \varphi_2)$	iff there is $\pi \in \Pi^{max}(s)$ and there is a position i in π s.t. $\pi_i \models \varphi_2$ and for all j, $0 \leq j < i$, we have $\pi_j \models \varphi_1$

Table 1: Semantics of CTL formulae

We can now define weighted Petri nets with inhibitor arcs. Let $\mathbb{N}^0 = \mathbb{N} \cup \{0\}$ be the set of natural numbers including 0 and let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ be the set of natural numbers including infinity.

Definition 1 (Petri net). *A Petri net is a tuple $N = (P, T, W, I)$ where P and T are finite disjoint sets of places and transitions, $W : (P \times T) \cup (T \times P) \to \mathbb{N}^0$ is the weight function for regular arcs, and $I : (P \times T) \to \mathbb{N}^\infty$ is the weight function for inhibitor arcs.*

A *marking* M on N is a function $M : P \to \mathbb{N}^0$ where $M(p)$ denotes the number of tokens in the place p. The set of all markings of a Petri net N is written as $\mathcal{M}(N)$. Let $M_0 \in \mathcal{M}(N)$ be a given *initial marking* of N.

A Petri net $N = (P, T, W, I)$ defines an LTS $TS(N) = (\mathcal{S}, A, \to)$ where $\mathcal{S} = \mathcal{M}(N)$ is the set of all markings, $A = T$ is the set of labels, and $M \xrightarrow{t} M'$ whenever for all $p \in P$ we have $M(p) < I((p,t))$ and $M(p) \geq W((p,t))$ such that $M'(p) = M(p) - W((p,t)) + W((t,p))$. We inductively extend the relation $\xrightarrow{t}$ to sequences of transitions $w \in T^*$ such that $M \xrightarrow{\epsilon} M$ and $M \xrightarrow{wt} M'$ if $M \xrightarrow{w} M''$ and $M'' \xrightarrow{t} M'$. We write $M \to^* M'$ if there is $w \in T^*$ such that $M \xrightarrow{w} M'$. By $\mathit{reach}(M) = \{M' \in \mathcal{M}(N) \mid M \to^* M'\}$ we denote the set of all markings reachable from M.

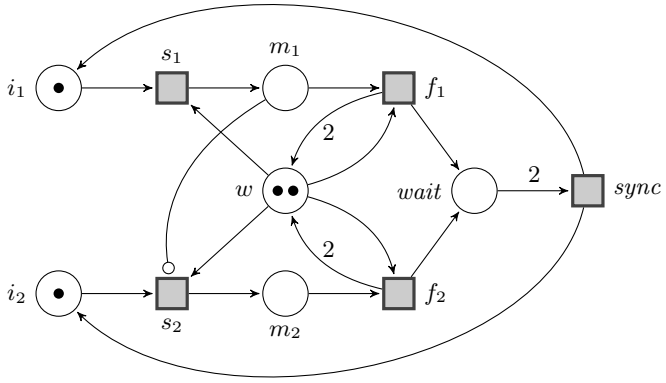

Fig. 1: A Petri net modelling two synchronizing processes

Example 1. Figure 1 illustrates an example of a Petri net where places are drawn as circles, transitions as rectangles, regular arcs as arrows with the weight as labels (default weight is 1 and arcs with weight 0 are not depicted) and inhibitor arcs are shown as circle-headed arrows (again the default weight is 1 and arcs with weight ∞ are not depicted). The dots inside places represent the number of tokens (marking). The initial marking in the net can be written by $i_1 i_3 2w$ denoting one token in i_1, one token in i_3 and two tokens in the place w. The net attempts to model two processes that aim to get exclusive access to firing either the transition f_1 or f_2 (making sure that they cannot be enabled concurrently). Once the first process decides to enable transition f_1 by moving the token from i_1 to m_1, the second process is not allowed to place a token into m_2 due to the inhibitor arc connection m_1 to s_2. However, as there is no inhibitor arc in the order direction, it is possible to reach a deadlock in the net by performing $i_1 i_2 2w \xrightarrow{s_2} i_1 m_2 w \xrightarrow{s_1} m_1 m_2$.

Finally, we fix the set of atomic propositions α ($\alpha \in AP$) for Petri nets as used in the MCC Property Language [15]:

$$\alpha ::= \ t \mid e_1 \bowtie e_2$$

$$e ::= \ c \mid p \mid e_1 \oplus e_2$$

where $t \in T$, $c \in \mathbb{N}^0$, $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $p \in P$, and $\oplus \in \{+, -, *\}$. The evaluation function v for a marking M is given as $v(M) = \{t \in T \mid t \in en(M)\} \cup \{e_1 \bowtie e_2 \mid eval_M(e_1) \bowtie eval_M(e_2)\}$ where $eval_M(c) = c$, $eval_M(p) = M(p)$ and $eval_M(e_1 \oplus e_2) = eval_M(e_1) \oplus eval_M(e_2)$.

Formulae that do not use any atomic predicate t for transition firing and *deadlock* are called *CTL cardinality formulae* and formulae that avoid the use of

$e_1 \bowtie e_2$ and *deadlock* are called *CTL firability formulate*. Formulae of the form $EF\varphi$ or $AG\varphi$ where φ does not contain any other temporal operator are called *reachability formulae*, and as for CTL can be subdivided into the *reachability cardinality* and *reachability fireability* category.

Example 2. Consider the Petri net in Figure 1 and the reachability fireability formula $EF(f_1 \wedge f_2)$ asking whether there is a reachable marking that enables both f_1 and f_2. By exploring the (finite) part of the LTS reachable from the initial marking $i_1 i_2 2w$, we can conclude that $i_1 i_2 2w \not\models EF(f_1 \wedge f_2)$. However, the slightly modified query $EFAX(f_1 \wedge f_2)$ holds in the initial marking as a deadlocked marking $m_1 m_2$ can be reached, and due to the definition of the universal next modality we have $m_1 m_2 \models AX(f_1 \wedge f_2)$. Another example of a cardinality formula is $E(w \geq 2 \ U \ m_2 = 1)$ asking if there is a computation that marks the place m_2 and before it happens, w must contain at least two tokens. This formula holds in the initial marking by firing the transition s_2.

We shall finish the preliminaries by recalling the basics of linear programming. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables and let $\overline{x} = (x_1, x_2, \ldots, x_n)^T$ be a column vector of the variables. A *linear equation* is of the form $\overline{c} \cdot \overline{x} \bowtie k$ where $\bowtie \in \{=, <, \leq, >, \geq\}$, $k \in \mathbb{Z}$ is an integer, and $\overline{c} = (c_1, c_2, \ldots, c_n)$ is a row vector of integer constants. An *integer linear program* LP is a finite set of linear equations. An (integer) *solution* to LP is a mapping $u : X \to \mathbb{N}^0$ from variables to natural numbers such that for every linear equation $(\overline{c} \cdot \overline{x} \bowtie k) \in LP$, the column vector $\overline{u} = (u(x_1)\ u(x_2) \cdots u(x_n))^T$ satisfies the equation $\overline{c} \cdot \overline{u} \bowtie k$. We use $\mathcal{E}^X_{lin}$ to denote the set of all integer linear programs over the variables X.

An integer linear program with a solution is said to be *feasible*. For our purpose, we only consider feasibility and we are not interested in the optimality of the solution. The feasibility problem of integer linear programs is NP-complete [11,19], however, there exists a number of efficient linear program solvers (we use `lp_solve` in our implementation [3]).

3 Logical Equivalence of Formulae

Before we give our method for recursive simplification of CTL formulae via the use of state equations in Section 4, we first introduce two other formula simplification techniques. The first method utilizes the initial marking and the second method uses universally valid formulae equivalences. For the rest of this section, we assume a fixed Petri net $N = (P, T, W, I)$ with the initial marking M_0.

For the first simplification, let us define in Table 2 the function $\Omega : \Phi_{CTL} \to \{true, false, ?\}$ that checks if a given formula is trivially satisfiable in the initial marking M_0. Note that we generalize the binary conjunctions and disjunctions to n-ary operations as it corresponds to the implementation in our tool. The correctness of this simplification is expressed in the following theorem.

Theorem 1 (Initial Rewrite). *Let φ be a CTL formula such that $\Omega(\varphi) \neq ?$. Then $M_0 \models \varphi$ if and only if $\Omega(\varphi) = true$.*

$$\Omega(\mathit{true}) = \mathit{true} \qquad \Omega(\mathit{false}) = \mathit{false}$$

$$\Omega(\alpha) = M_0 \models \alpha \qquad \Omega(\mathit{deadlock}) = M_0 \models \mathit{deadlock}$$

$$\Omega(AX\varphi) = \begin{cases} \mathit{true} & \text{if } M_0 \models \mathit{deadlock} \\ ? & \text{otherwise} \end{cases} \qquad \Omega(EX\varphi) = \begin{cases} \mathit{false} & \text{if } M_0 \models \mathit{deadlock} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(\neg\varphi) = \begin{cases} \mathit{true} & \text{if } \Omega(\varphi) = \mathit{false} \\ \mathit{false} & \text{if } \Omega(\varphi) = \mathit{true} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(\varphi_1 \wedge \cdots \wedge \varphi_n) = \begin{cases} \mathit{true} & \text{if for all } i,\ 1 \leq i \leq n, \text{ we have } \Omega(\varphi_i) = \mathit{true} \\ \mathit{false} & \text{if there exists } i,\ 1 \leq i \leq n, \text{ s.t. } \Omega(\varphi_i) = \mathit{false} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(\varphi_1 \vee \cdots \vee \varphi_n) = \begin{cases} \mathit{true} & \text{if there exists } i,\ 1 \leq i \leq n, \text{ s.t. } \Omega(\varphi_i) = \mathit{true} \\ \mathit{false} & \text{if for all } i,\ 1 \leq i \leq n, \text{ we have } \Omega(\varphi_i) = \mathit{false} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(EG\varphi) = \Omega(AG\varphi) = \begin{cases} \mathit{false} & \text{if } \Omega(\varphi) = \mathit{false} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(EF\varphi) = \Omega(AF\varphi) = \begin{cases} \mathit{true} & \text{if } \Omega(\varphi) = \mathit{true} \\ ? & \text{otherwise} \end{cases}$$

$$\Omega(E(\varphi_1 U \varphi_2)) = \Omega(A(\varphi_1 U \varphi_2)) = \begin{cases} \mathit{true} & \text{if } \Omega(\varphi_2) = \mathit{true} \\ \mathit{false} & \text{if } \Omega(\varphi_1) = \Omega(\varphi_2) = \mathit{false} \\ ? & \text{otherwise} \end{cases}$$

Table 2: Simplification rules for a given initial marking M_0

For the second simplification, we establish a recursively defined rewrite-function $\rho : \Phi_{CTL} \to \Phi_{CTL}$ given in Table 3 that is based on logical equivalences for the CTL quantifiers. In the definition of ρ, we assume that the n-ary operators $\vee$ and $\wedge$ are associative and commutative. The correctness is captured in the following theorem.

Theorem 2 (Equivalence Rewriting). *Let $M \in \mathcal{M}(N)$ be a marking on N. Then $M \models \varphi$ if and only if $M \models \rho(\varphi)$.*

4 Formula Simplification via State Equations

We will now describe the main ingredients of our formula simplification algorithm. It is based on a recursive decent on the structure of the formula, checking whether its subformulae and their negations can possibly hold in some reachable

$$\rho(\alpha) = \alpha \qquad \rho(deadlock) = deadlock$$
$$\rho(EG\varphi) = \rho(\neg AF\rho(\neg\varphi)) \qquad \rho(AG\varphi) = \rho(\neg EF\rho(\neg\varphi))$$
$$\rho(EX\varphi) = EX\rho(\varphi) \qquad \rho(AX\varphi) = AX\rho(\varphi)$$
$$\rho(\varphi_1 \wedge \cdots \wedge \varphi_n) = \rho(\varphi_1) \wedge \cdots \wedge \rho(\varphi_n) \qquad \rho(\varphi_1 \vee \cdots \vee \varphi_n) = \rho(\varphi_1) \vee \cdots \vee \rho(\varphi_n)$$

$$\rho(\neg\varphi) = \begin{cases} \varphi' & \text{if } \rho(\varphi) = \neg\varphi' \\ AX\rho(\neg\varphi') & \text{if } \rho(\varphi) = EX\varphi' \\ EX\rho(\neg\varphi') & \text{if } \rho(\varphi) = AX\varphi' \\ \rho((\neg\varphi_1) \wedge \cdots \wedge (\neg\varphi_n)) & \text{if } \varphi = \varphi_1 \vee \cdots \vee \varphi_n \\ \rho((\neg\varphi_1) \vee \cdots \vee (\neg\varphi_n)) & \text{if } \varphi = \varphi_1 \wedge \cdots \wedge \varphi_n \\ \neg\rho(\varphi) & \text{otherwise} \end{cases}$$

$$\rho(EF\varphi) = \begin{cases} \neg deadlock & \text{if } \rho(\varphi) = \neg deadlock \\ EF\varphi' & \text{if } \rho(\varphi) = EF\varphi' \\ \rho(EF\varphi') & \text{if } \rho(\varphi) = AF\varphi' \\ \rho(EF\varphi_2) & \text{if } \rho(\varphi) = E(\varphi_1 U \varphi_2) \\ \rho(EF\varphi_2) & \text{if } \rho(\varphi) = A(\varphi_1 U \varphi_2) \\ \rho(EF\varphi_1 \vee \cdots \vee EF\varphi_n) & \text{if } \rho(\varphi) = \varphi_1 \vee \cdots \vee \varphi_n \\ EF\rho(\varphi) & \text{otherwise} \end{cases}$$

$$\rho(AF\varphi) = \begin{cases} \neg deadlock & \text{if } \rho(\varphi) = \neg deadlock \\ EF\varphi' & \text{if } \rho(\varphi) = EF\varphi' \\ AF\varphi' & \text{if } \rho(\varphi) = AF\varphi' \\ \rho(AF\varphi_2) & \text{if } \rho(\varphi) = A(\varphi_1 U \varphi_2) \\ \rho((EF\varphi_2) \vee (AF\varphi_1)) & \text{if } \rho(\varphi) = \varphi_1 \vee EF\varphi_2 \\ AF\rho(\varphi) & \text{otherwise} \end{cases}$$

$$\rho(A(\varphi_1 U \varphi_2)) = \begin{cases} \neg deadlock & \text{if } \rho(\varphi_2) = \neg deadlock \\ \rho(\varphi_2) & \text{if } \rho(\varphi_1) = deadlock \\ \rho(AF\varphi_2) & \text{if } \rho(\varphi_1) = \neg deadlock \\ EF\varphi_3 & \text{if } \rho(\varphi_2) = EF\varphi_3 \\ AF\varphi_3 & \text{if } \rho(\varphi_2) = AF\varphi_3 \\ \rho((EF\varphi_4) \vee A(\varphi_1 U \varphi_3)) & \text{if } \rho(\varphi_2) = \varphi_3 \vee EF\varphi_4 \\ A(\rho(\varphi_1) U \rho(\varphi_2)) & \text{otherwise} \end{cases}$$

$$\rho(E(\varphi_1 U \varphi_2)) = \begin{cases} \neg deadlock & \text{if } \rho(\varphi_2) = \neg deadlock \\ \rho(\varphi_2) & \text{if } \rho(\varphi_1) = deadlock \\ \rho(EF\varphi_2) & \text{if } \rho(\varphi_1) = \neg deadlock \\ EF\varphi_3 & \text{if } \rho(\varphi_2) = EF\varphi_3 \\ \rho((EF\varphi_4) \vee E(\varphi_1 U \varphi_3) & \text{if } \rho(\varphi_2) = \varphi_3 \vee EF\varphi_4 \\ E(\rho(\varphi_1) U \rho(\varphi_2)) & \text{otherwise} \end{cases}$$

Table 3: Equivalence rewriting of CTL formulae

marking (here we use the state equation [11,17] approach) and then propagating back this information through the Boolean and temporal operators.

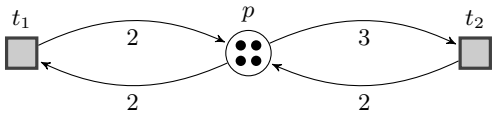

Fig. 2: Example Petri net and initial marking for formula simplification

We use state equations to identify universally true or false subformulae, similarly as e.g. in [14]. The main novelty is that we extend the approach to deal with arbitrary arithmetical expressions and repeatedly solve linear programs for subformulae of the given property so that more significant simplifications can be achieved (we try to solve the state equations both for the subformula and its negation). As a result, we can simplify more formulae into the trivially valid ones (*true*) or invalid ones (*false*) or we can significantly reduce the size of the formulae which can then speed up the state space exploration.

Consider the Petri net in Figure 2 with the initial marking M_0, where $M_0(p) = 4$. The state equation for the reachability formula $EF\ p \geq 5$ (can the place p be marked with at least five tokens) over the variables x_{t_1} and x_{t_2} (representing the number of transition firings of t_1 and t_2 respectively) looks as

$$M_0(p) + \sum_{t \in T} (W(t,p) - W(p,t)) x_t \geq 5$$

which in our example translates to $4 + 0 \cdot x_{t_1} - 1 \cdot x_{t_2} \geq 5$. The inequality clearly does not have a solution in nonnegative integers, hence we can conclude without exploring the state space that $EF\ p \geq 5$ does not hold in the initial marking. Moreover, consider now the formula $EF\ (p \geq 5) \vee (p = 2 \wedge p \leq 7)$. By recursively analyzing the subformulae, we can conclude using the state equations that $p \geq 5$ cannot be satisfied in any reachable marking, hence the formula simplifies to $EF\ (p = 2 \wedge p \leq 7)$. Moreover, by continuing the recursive decent and looking at the subformula $p \leq 7$, we can determine by using state equations, that its negation $p > 7$ cannot be satisfied in any reachable marking. Hence $p \leq 7$ is universally true and the formula further simplifies to an equivalent formula $EF\ p = 2$ for which we have to apply conventional verification techniques.

In what follows, we formally define our formula simplification procedure and extend it to the full CTL logic so that e.g. the formula $EF\ AX\ p \geq 5$ simplifies to the reachability formula *EF deadlock* for which we can use specialized algorithms for deadlock detection (e.g. using the siphon-trap property [13]) instead of the more expensive CTL verification algorithms. Even if a CTL formula does not simplify to a pure reachability property, the reduction in the size of the CTL formula has still a positive effect on the efficiency of the CTL verification algorithms as the state space grows with the number of different subformulae.

φ	rewritten φ
t	$p_1 \geq W(p_1,t) \wedge \cdots \wedge p_n \geq W(p_n,t) \wedge$ $p_1 < I(p_1,t) \wedge \cdots \wedge p_n < I(p_n,t)$ where $P = \{p_1, p_2, \ldots, p_n\}$
$e_1 \neq e_2$	$e_1 > e_2 \vee e_1 < e_2$
$e_1 = e_2$	$e_1 \leq e_2 \wedge e_1 \geq e_2$
$\neg(\varphi_1 \wedge \varphi_2)$	$\neg\varphi_1 \vee \neg\varphi_2$
$\neg(\varphi_1 \vee \varphi_2)$	$\neg\varphi_1 \wedge \neg\varphi_2$
$\neg AX\varphi$	$EX\neg\varphi$
$\neg EX\varphi$	$AX\neg\varphi$
$\neg AF\varphi$	$EG\neg\varphi$
$\neg EF\varphi$	$AG\neg\varphi$
$\neg AG\varphi$	$EF\neg\varphi$
$\neg EG\varphi$	$AF\neg\varphi$

Table 4: Rewriting rules

4.1 Simplification Procedure

Let $N = (P, T, W, I)$ be a fixed Petri net with the initial marking M_0 and φ a given CTL formula. Before we start, we assume that the formula φ has been rewritten into an equivalent one by recursively applying the rewriting rules in Table 4. Clearly, these rules preserve logical equivalence and they push the negation down to either the atomic propositions or in front of the existential or universal until operators. Moreover, the fireability predicate for a transition t is rewritten to the equivalent cardinality formula.

Let $\mathcal{E}^X_{lin}$ be the set of all integer linear programs over the set of variables $X = \{x_t \mid t \in T\}$. Let $LPS \subseteq \mathcal{E}^X_{lin}$ be a finite set of integer linear programs. We say that LPS has a solution, if there exists a linear program $LP \in LPS$ that has a solution.

We will now define a simplification function that, for a given formula $\varphi \in \Phi_{CTL}$, produces a simplified formula and two sets of integer linear programs. The function is of the form

$$\mathit{simplify} : \Phi_{CTL} \to \Phi_{CTL} \times 2^{\mathcal{E}^X_{lin}} \times 2^{\mathcal{E}^X_{lin}}$$

and we write $\mathit{simplify}(\varphi) = (\varphi', LPS, \overline{LPS})$ when the formula φ is simplified to an equivalent formula φ', and where the following invariant holds:

- if $M \models \varphi$ for some M reachable from M_0 then LPS has a solution, and
- if $M \not\models \varphi$ for some M reachable from M_0 then $\overline{LPS}$ has a solution.

In order to define the simplification function, we use the function $\mathit{merge} : 2^{\mathcal{E}^X_{lin}} \times 2^{\mathcal{E}^X_{lin}} \to 2^{\mathcal{E}^X_{lin}}$ that combines two set of integer linear programs and is defined as $\mathit{merge}\ (LPS_1, LPS_2) = \{LP_1 \cup LP_2 \mid LP_1 \in LPS_1,\ LP_2 \in LPS_2\}$. Finally, let $BASE$ denote the integer linear program with the following equations

$$M_0(p) + \sum_{t \in T} (W(t,p) - W(p,t)) \cdot x_t \geq 0 \quad \text{for all } p \in P$$

Algorithm 1: Simplify $e_1 \bowtie e_2$

Function *simplify*$(e_1 \bowtie e_2)$
  if e_1 *is not linear or* e_2 *is not linear* **then**
    return $(e_1 \bowtie e_2, \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$
  $LPS \leftarrow \{\{const(e_1) \bowtie const(e_2)\}\}$
  $\overline{LPS} \leftarrow \{\{const(e_1) \mathbin{\overline{\bowtie}} const(e_2)\}\}$
  if $\{LP \cup BASE \mid LP \in LPS\}$ *has no solution* **then**
    return *simplify*(*false*)
  else if $\{LP \cup BASE \mid LP \in \overline{LPS}\}$ *has no solution* **then**
    return *simplify*(*true*)
  else
    return $(e_1 \bowtie e_2, LPS, \overline{LPS})$

Algorithm 2: Simplify $\neg\varphi$

Function *simplify*$(\neg\varphi)$
  $(\varphi', LPS, \overline{LPS}) \leftarrow$ *simplify*(φ)
  if $\varphi' =$ *true* **then return** *simplify*(*false*)
  if $\varphi' =$ *false* **then return** *simplify*(*true*)
  return $(\neg\varphi', \overline{LPS}, LPS)$

that ensures that any solution to $BASE$ must leave a nonnegative number of tokens in every place of N.

First, we postulate $simplify(true) = (true, \{\{0 \leq 1\}\}, \emptyset)$, $simplify(false) = (false, \emptyset, \{\{0 \leq 1\}\})$, and $simplify(deadlock) = (deadlock, \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$ and these definitions clearly satisfy our invariant.

Algorithm 1 describes how to simplify the atomic predicates, where the function *const* takes as input an arithmetic expression e and returns one side of the linear equation as follows:

$$\begin{aligned}
const(c) &= c \\
const(p) &= M_0(p) + \sum_{t \in T} (W(t,p) - W(p,t)) \cdot x_t \\
const(e_1 + e_2) &= const(e_1) + const(e_2) \\
const(e_1 - e_2) &= const(e_1) - const(e_2) \\
const(e_1 \cdot e_2) &= const(e_1) \cdot const(e_2).
\end{aligned}$$

In the algorithm we let $\overline{\bowtie}$ denote the dual operation to $\bowtie$, for example $\overline{>}$ becomes $\leq$ and $\overline{\geq}$ becomes $<$. There is a special case that we must handle here. If in either of the expressions e_1 or e_2 we have a multiplication that includes more than one place (i.e. the expression is not linear) then we would return a nonlinear program that cannot be solved by linear program solvers. To handle this situation, if either side of the comparison in nonlinear, we return the formula unchanged and two

Algorithm 3: Simplify $\varphi_1 \Diamond \ldots \Diamond \varphi_n$ for $\Diamond \in \{\wedge, \vee\}$

```
1  Function simplify(φ_1 ◇ ... ◇ φ_n)
2      Let φ' be an empty formula.
3      if ◇ = ∧ then  LPS ← {{0 ≤ 1}};  overline(LPS) ← ∅
4      if ◇ = ∨ then  LPS ← ∅;  overline(LPS) ← {{0 ≤ 1}}
5      for i := 1 to n do
6          (φ'_i, LPS_i, overline(LPS)_i) ← simplify(φ_i)
7          if ◇ = ∧ and φ'_i = false then return simplify(false)
8          if ◇ = ∧ and φ'_i ≠ true then
9              φ' ← φ' ∧ φ'_i
10             LPS ← merge(LPS, LPS_i)
11             overline(LPS) ← overline(LPS) ∪ overline(LPS)_i
12         if ◇ = ∨ and φ'_i = true then return simplify(true)
13         if ◇ = ∨ and φ'_i ≠ false then
14             φ' ← φ' ∨ φ'_i
15             LPS ← LPS ∪ LPS_i
16             overline(LPS) ← merge(overline(LPS), overline(LPS)_i)
17     if φ' is empty formula and ◇ = ∧ then return simplify(true)
18     if φ' is empty formula and ◇ = ∨ then return simplify(false)
19     if ◇ = ∧ and {LP ∪ BASE | LP ∈ LPS} has no solution then
20         return simplify(false)
21     if ◇ = ∨ and {LP ∪ BASE | LP ∈ overline(LPS)} has no solution then
22         return simplify(true)
23     return (φ', LPS, overline(LPS))
```

singleton sets of linear programs $\{\{0 \leq 1\}\}$ that trivially have a solution (any variable assignment is a solution to the linear program $0 \leq 1$) and hence satisfy our invariant.

The simplification of negation $\neg\varphi$ is given in Algorithm 2. It first recursively computes the simplification φ' of φ and if the answer is conclusive then the negated conclusive answer is returned, otherwise we return $\neg\varphi'$ and swap the two sets of linear programs.

In Algorithm 3 we show how to simplify conjunctions and disjunctions of formulae. We give the simplification function for n-ary operators to mimic the implementation closely. We present both conjunction and disjunction in the same pseudocode in order to clarify the symmetry in handling the Boolean connectives. The algorithm recursively simplifies the subformulae and one by one adds the simplified formulae into the resulting proposition φ', unless a conclusive answer (*true*/*false*) can be given immediately or the subformula can be omitted. Note that for conjunction we merge the current LPS and LPS_i returned for the subformula φ_i as if the conjunction is satisfied in some reachable marking then there must be an $LP \in LPS$ and an $LP_i \in LPS_i$ such that $LP \cup LP_i$ has a solution. Symmetrically, we do the merge also for disjunction and the negated

Algorithm 4: Simplify $QX\varphi$, where $Q \in \{A, E\}$

```
1 Function simplify(QXφ)
2     (φ', LPS, LPS̄) ← simplify(φ)
3     if Q = A and φ' = true then return simplify(true)
4     if Q = A and φ' = false then return simplify(deadlock)
5     if Q = E and φ' = true then return simplify(¬deadlock)
6     if Q = E and φ' = false then return simplify(false)
7     return (QXφ', {{0 ≤ 1}}, {{0 ≤ 1}})
```

Algorithm 5: Simplify $QP\varphi$, where $QP \in \{AG, EG, AF, EF\}$

```
1 Function simplify(QPφ)
2     (φ', LPS, LPS̄) ← simplify(φ)
3     if φ' = true then return simplify(true)
4     if φ' = false then return simplify(false)
5     return (QPφ', {{0 ≤ 1}}, {{0 ≤ 1}})
```

sets of linear programs. Finally, we check whether the created systems of linear programs have solutions and in the negative cases we can sometimes draw a conclusive answer.

Simplification of the next operators is given in Algorithm 4. It is worth noticing that for certain situations, the next operator can be removed and replaced with the deadlock proposition (and hence possibly change the CTL formula into a reachability formula). If none of the simplification cases applies, we return the next operator with the simplified formula together with two sets of linear programs with trivial solutions in order to satisfy our invariant. Similarly, the simplification of the unary CTL temporal operators is given in Algorithm 5.

Finally, in Algorithm 6 we present the simplification of binary CTL temporal operators. Here we first simplify φ_2 and see if we can draw some straightforward conclusions. If this is not the case, we also simplify φ_1 and if it evaluates to *true* or *false*, we can either reduce the binary temporal operator into a unary one or completely remove the unary operator, respectively.

Example 3. Consider again the net from Example 2. We can simplify the formula $EFAX(f_1 \wedge f_2)$ as follows. Let $X = \{x_{s_1}, x_{s_2}, x_{f_1}, x_{f_2}, x_{sync}\}$ be the variables. Using the rewriting rules from Table 4 we have that $EFAX(f_1 \wedge f_2)$ is equivalent to $EFAX(m_1 \geq 1 \wedge w \geq 1 \wedge m_2 \geq 1)$. The linear equations LPS generated by Algorithm 1 and 3 are as follows.

$$\begin{aligned} x_{s_1} - x_{f_1} &\geq 1 \\ 2 + x_{f_1} + x_{f_2} - x_{s_1} - x_{s_2} &\geq 1 \\ x_{s_2} - x_{f_2} &\geq 1 \end{aligned}$$

We do not include $BASE$ here, as the equations above are already unfeasible (have no integer solution). This follows from the observation that the first and

Algorithm 6: Simplify $Q(\varphi_1 U \varphi_2)$, where $Q \in \{A, E\}$

Function *simplify*$(Q(\varphi_1 U \varphi_2))$
  $(\varphi_2', LPS_2, \overline{LPS}_2) \leftarrow$ *simplify*(φ_2)
  if $\varphi_2' =$ *true* **then return** *simplify*(*true*)
  if $\varphi_2' =$ *false* **then return** *simplify*(*false*)
  $(\varphi_1', LPS_1, \overline{LPS}_1) \leftarrow$ *simplify*(φ_1)
  if $\varphi_1' =$ *true* **then return** $(QF\varphi_2', \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$
  if $\varphi_1' =$ *false* **then return** $(\varphi_2', LPS_2, \overline{LPS}_2)$
  return $(Q(\varphi_1' U \varphi_2'), \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$

third equation imply that $x_{s_1} > x_{f_1}$ and $x_{s_2} > x_{f_2}$, respectively, and this contradicts the second equation $2 + x_{f_1} + x_{f_2} > x_{s_1} + x_{s_2}$. Therefore, Algorithm 3 simplifies $EFAX(f_1 \wedge f_2)$ to $EFAXfalse$ and by Algorithm 4, we simplify it further to $EFdeadlock$. No further reduction is possible, however, we simplified a CTL formula into a simple reachability formula for which we can now use specialized algorithms for deadlock detection.

We conclude this section with a theorem stating the correctness of the simplification, meaning that for $simplify(\varphi) = (\varphi', LPS, \overline{LPS})$ we have $M_0 \models \varphi$ if and only if $M_0 \models \varphi'$. In order to do so, we prove a stronger claim that allows us to formally introduce the invariant on the sets of linear programs returned by the function *simplify*.

Theorem 3 (Formula Simplification Correctness). *Let $N = (P, T, W, I)$ be a Petri net, M_0 an initial marking on N, and $\varphi \in \Phi_{CTL}$ a CTL formula. Let $simplify(\varphi) = (\varphi', LPS, \overline{LPS})$. Then for all markings $M \in \mathcal{M}(N)$ such that $M_0 \xrightarrow{w} M$ holds:*

1. *$M \models \varphi$ iff $M \models \varphi'$*
2. *if $M \models \varphi$ then there is $LP \in LPS$ such that $\wp(w)$ is a solution to LP*
3. *if $M \not\models \varphi$ then there is $LP \in \overline{LPS}$ such that $\wp(w)$ is a solution to LP*

where $\wp(w)$ is a solution that assigns to each variable x_t the number of occurrences of the transition t in the transition sequence w.

5 Implementation and Experiments

The formula simplification techniques are implemented in C++ in the `verifypn` engine [14] of the tool TAPAAL [10] and distributed in the latest release at `www.tapaal.net`. The source code is available at `code.launchpad.net/verifypn`.

After parsing the PNML model and the formula, TAPAAL applies sequentially the simplification procedures as depicted in Figure 3, where we first attempt to restructure the formulae to a simpler form using ρ followed by the application of Ω. After this, the main *simplify* procedure is called. The simplification can create a formula where additional applications of ρ and Ω are possible

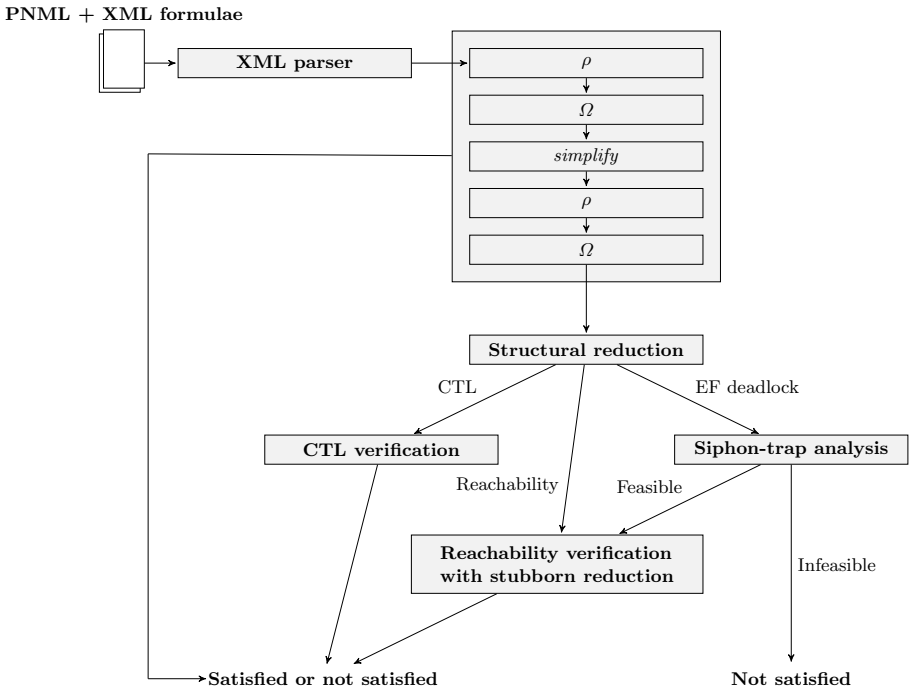

Fig. 3: TAPAAL tool-chain and control flow

and can further reduce the formula size. After the simplification is completed, TAPAAL applies structural reductions to the model, removing or merging redundant transitions and places as described in [14]. The engine now proceeds as follows.

1. If the formulae is of the form $EF\,deadlock$ then siphon-trap analysis is attempted, followed by normal explicit-state verification in case of an inconclusive answer.
2. If the formulae falls within pure reachability category ($EF\varphi$ or $\neg EF\varphi$, where φ does not contain further temporal operators), then we call a specialized reachability engine that uses stubborn set reduction.
3. For the general CTL formula, the verification is performed via a translation to a dependency graph and performing on-the-fly computation of its minimum fixed-point assignment as described in [9].

5.1 Implementation Details of the Simplification Procedure

During implementation and subsequent experimentation, we discovered that the construction of linear programs for large models can be both time and memory-consuming. In particular, the *merge*-operation causes a quadratic blowup both in the size and the number of linear programs. To remedy this, we have implemented

CTL Cardinality

Algorithm	Solved	% Solved	Reachability	% Reachability	% Reduction
Ω	117	2.3	1834	36.6	27.2
ρ	7	0.1	1437	28.7	24.1
simplify	1437	28.7	2425	48.4	45.7
all	**1724**	**34.4**	**2993**	**59.8**	**60.3**

CTL Fireability

Ω	194	3.9	1701	34.0	27.1
ρ	0	0.0	1319	26.3	30.0
simplify	255	5.1	1422	28.4	11.0
all	**495**	**9.9**	**2022**	**40.4**	**49.7**

Table 5: Formula simplification for CTL cardinality and fireability

a "lazy" construction of the linear programs—similar to lazy evaluation known from functional programming languages. Instead of computing the full set of linear programs up front, we simply remember the basic linear programs and the tree of operations making up the merged or unioned linear program. Using this construction, we then extract a single linear program on demand, and thus avoid the up-front time and memory overhead of computing the merge and union operations at the call time.

5.2 Experimental Setup

To evaluate the performance of our approach, we conduct two series of experiments on the models and formulae from MCC'17 [15]. First, we investigate the effect of the three different simplification methods proposed in this paper along with their combination as depicted in Figure 3. In the second experiment we compare the performance of our simplification algorithms to those used by LoLA, the winner of MCC'17. We also conduct a full run of the verification engines after the formula simplification in order to assess the impact of the simplification on the state space search. All experiments were run on AMD Opteron 6376 Processors, restricted to 14 GB of memory on 313 P/T nets from the MCC'17 benchmark. Each category contains 16 different queries which yields a total of 5008 executions for a given category.

5.3 Evaluation of Formula Simplification Techniques

We compare the performance of Ω, ρ and *simplify* functions along with their combined version referred to as *all* (applying sequentially ρ, Ω, *simplify*, ρ and Ω). The execution of each simplification was limited to 20 minutes per formula (excluding the model parsing time) and a timeout for finding a solution to a linear program using `lp_solve` [3] was set to 120 seconds.

CTL Simplification Only

	TAPAAL		**LoLA**		**LoLA+Sara**	
	Solved	% Solved	Solved	% Solved	Solved	% Solved
Cardinality	1724	34	236	5	904	18
Fireability	495	10	173	3	488	10
Total	**2219**	**22**	**409**	**4**	**1392**	**14**

CTL Simplification Followed by Verification

Cardinality	4232	85	3634	73	3810	76
Fireability	3712	74	3663	73	3690	74
Total	**7944**	**79**	**7297**	**73**	**7500**	**75**

Table 6: Tool comparison on CTL formulae

Table 5 reports the numbers (and percentages) of formulae that were solved (simplified to either *true* or *false*), the number of formulae converted from a complex CTL formula into a pure reachability formula and the average formula reduction in percentages (where the formula size before and after the reductions is measured as a number of nodes in its parse tree).

We can observe that the combination of our techniques simplifies about 34% of cardinality queries and 10% of fireability queries into *true* or *false*, while a significant number of queries are simplified from CTL formula into pure reachability problems (60% of cardinality queries and 40% of fireability ones). The average reduction in the query size is 60% for cardinality and 50% for fireability queries. The results are encouraging, though the performance on fireability formulae is considerably worse than for cardinality formulae. The reason is that fireability predicates are translated into Boolean combinations of cardinality predicates and the expanded formulae are less suitable for the simplification procedures due to their increased size. This is also reflected by the time it took to compute the simplification. For CTL cardinality, half of the simplifications terminate in less than 0.05 seconds, 75% simplifications terminate in less than 0.98 seconds and 90% of simplifications terminate in less than 9.46 seconds. The corresponding numbers for CTL fireability are 0.22 seconds, 13.70 seconds and 538.34 seconds.

5.4 Comparison with LoLA

We compare the performance of our tool-chain, presented in Figure 3, with the tool LoLA [26] and the combination of LoLA and its linear program solver Sara [25] that uses the CEGAR approach. In CTL simplification experiment, we allow 20 minutes for formula simplification (excluding the net parsing time) and count how many solved (simplified to *true* or *false*) queries each tool computed[1]. For CTL verification, we allow the tools first simplify the query and then proceed

[1] We use the current development snapshots of LoLA (based on version 2.0) and Sara (based on version 1.14), kindly provided by the LoLA and Sara development team.

Reachability Simplification Only

	TAPAAL		**LoLA**		**LoLA+Sara**	
	Solved	% Solved	Solved	% Solved	Solved	% Solved
Cardinality	2256	45	277	6	3734	75
Fireability	1073	21	296	6	2880	58
Total	**3329**	**33**	**573**	**6**	**6614**	**66**

Reachability Simplification Followed by Verification

Cardinality	4638	93	3734	75	4628	92
Fireability	4402	88	2880	58	4372	87
Total	**9040**	**90**	**6614**	**66**	**9000**	**90**

Table 7: Tool comparison on reachability formulae

with the verification according to the best setup the tools provide, again with a 20 minute timeout excluding the parsing time. We run LoLA and Sara in parallel (in their advantage), each of them having 20 minute timeout per execution. The results are presented in Table 6. We can observe that in simplification of CTL cardinality formulae, we are able to provide an answer for 34% of queries while the combination of LoLA and Sara solves only 18% of them. The performance on the CTL fireability simplification is comparable. If we follow the simplification with an actual verification, TAPAAL solves in total 79% of queries and LoLA with Sara 75%. As a result, TAPAAL with the new query simplification algorithms now outperforms the CTL category winner of the last year.

For completeness, in Table 7, we also include the results for the simplification and verification of reachability queries, even though our method is mainly targeted towards CTL formulae. We can notice that thanks to Sara, a fully functional model checker implementing the CEGAR approach, LoLA in combination with Sara solves twice as many queries by simplification as we do. However, once followed by the actual verification (and due to our simplification technique that significantly reduces formula sizes), both tools now show essentially comparable performance with a small margin towards TAPAAL, solving 40 additional formulae.

6 Conclusion

We presented techniques for reducing the size of a CTL formula interpreted over the Petri net model. The motivation is to speed up the state space search and to provide a beneficial interplay with other techniques like partial order and structural reductions. The experiential results—compared with LoLA, the winner of MCC'17 competition—document a convincing performance for simplification of CTL formulae as well as for CTL verification. The techniques were not designed specifically for the simplification of reachability formulae, hence the number of solved reachability queries by employing only the simplification is much lower

than that by the specialized tools like Sara (being in fact a complete model checker). However, when combined with the state space search followed after the formula simplification, the benefits of our techniques become apparent as we now solve 40 additional formulae compared to the combined performance of LoLA and Sara.

The simplification procedure is less efficient for CTL fireability queries than for CTL cardinality queries. This is the case both for our tool as well as LoLA and Sara. The reason is that we do not handle fireability predicates directly and unfold them into Boolean combination of cardinality predicates. This often results in significant explosion in query sizes. The future work will focus on overcoming this limitation and possibly handling the fireability predicates directly in the engine.

Acknowledgements. We would like to thank Karsten Wolf and Torsten Liebke from Rostock University for providing us with the development snapshot of the latest version of LoLA and for their help with setting up the tool and answering our questions. The last author is partially affiliated with FI MU, Brno.

References

1. G.S. Avrunin, U.A. Buy, and J.C. Corbett. Integer programming in the analysis of concurrent systems. In *International Conference on Computer Aided Verification (CAV'91)*, volume 575 of LNCS, pages 92–102. Springer, 1991.
2. G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, 1991.
3. M. Berkelaar, K. Eikland, P. Notebaert, et al. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.
4. M. Blondin, A. Finkel, C. Haase, and S. Haddad. Approaching the Coverability Problem Continuously. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *LNCS*, pages 480–496. Springer, 2016.
5. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.
6. E.M. Clarke, E.A Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.
8. J.C. Corbett and G.S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, 1995.
9. A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. Extended dependency graphs and efficient distributed fixed-point computation. In *Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'17)*, volume 10258 of *LNCS*, pages 139–158. Springer-Verlag, 2017.

10. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 492–497. Springer, 2012.
11. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16(2):159–189, 2000.
12. T. Geffroy, J. Leroux, and G. Sutre. Occam's Razor Applied to the Petri Net Coverability Problem. In *Reachability Problems*, volume 9899 of *LNCS*, pages 77–89. Springer, 2016.
13. M.H.T. Hack. Analysis of production schemata by Petri nets. Technical report, DTIC Document, 1972.
14. J.F. Jensen, T. Nielsen, L.K. Oestergaard, and J. Srba. TAPAAL and Reachability Analysis of P/T Nets. In *Transactions on Petri Nets and Other Models of Concurrency XI*, volume 9930 of *LNCS*, pages 307–318. Springer, 2016.
15. F. Kordon, H. Garavel, L.M. Hillah, F. Hulin-Hubard, B. Berthomieu, G. Ciardo, M. Colange, S. Dal Zilio, E. Amparore, M. Beccuti, T. Liebke, J. Meijer, A. Miner, C. Rohr, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf. Complete Results for the 2017 Edition of the Model Checking Contest. http://mcc.lip6.fr/2017/results.php, June 2017.
16. L.M. Kristensen, K. Schmidt, and A. Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design*, 29(3):215–251, 2006.
17. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
18. T. Murata and J.Y. Koh. Reduction and expansion of live and safe marked graphs. *IEEE Transactions on Circuits and Systems*, 27(1):68–70, 1980.
19. G.L. Nemhauser and L.A. Wolsey. Integer programming and combinatorial optimization. *Wiley, Chichester. GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, 20:8–12, 1988.
20. K. Schmidt. Stubborn sets for standard properties. In *International Conference on Application and Theory of Petri nets*, volume 1639 of *LNCS*, pages 46–65. Springer, 1999.
21. K. Schmidt. Integrating low level symmetries into reachability analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 315–330. Springer, 2000.
22. K. Schmidt. Narrowing Petri net state spaces using the state equation. *Fundamenta Informaticae*, 47(3-4):325–335, 2001.
23. A. Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer, 1989.
24. A. Valmari and H. Hansen. Stubborn set intuition explained. In *Transactions on Petri Nets and Other Models of Concurrency XII*, volume 10470 of *LNCS*, pages 140–165. Springer, 2017.
25. H. Wimmel and K. Wolf. Applying CEGAR to the Petri net state equation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*, pages 224–238. Springer, 2011.
26. K. Wolf. Running LoLA 2.0 in a Model Checking Competition. In *Transactions on Petri Nets and Other Models of Concurrency XI*, volume 9930 of *LNCS*, pages 274–285. Springer, 2016.

A Proof of Theorem 1

Proof. The proof proceeds by structural induction on φ.

- $\varphi = \mathit{true}$: Trivial.
- $\varphi = \mathit{false}$: Trivial.
- $\varphi = \alpha$: Trivial.
- $\varphi = \mathit{deadlock}$: Trivial.
- $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$: For all $i \in \{1, \ldots, n\}$, we know by the structural induction hypothesis that if $\Omega(\varphi_i) \neq ?$ then $M_0 \models \varphi_i$ iff $\Omega(\varphi_i) = \mathit{true}$. Assume that $\Omega(\varphi) \neq ?$ is true. We need to show the following two implications: (1) if $M_0 \models \varphi_1 \wedge \cdots \wedge \varphi_n$ then $\Omega(\varphi_1 \wedge \cdots \wedge \varphi_n) = \mathit{true}$, and (2) if $\Omega(\varphi_1 \wedge \cdots \wedge \varphi_n) = \mathit{true}$ then $M_0 \models \varphi_1 \wedge \cdots \wedge \varphi_n$.
  - Implication (1): Assume $M_0 \models \varphi_1 \wedge \cdots \wedge \varphi_n$. Then for all $i \in \{1, \ldots, n\}$ we must have that $M_0 \models \varphi_i$. Assume for the sake of contradiction that there exists $i \in \{1, \ldots, n\}$ s.t. $\Omega(\varphi_i) = \mathit{false}$. Then by the induction hypothesis we have $M_0 \not\models \varphi_i$, which is a contradiction. Therefore we must have $\Omega(\varphi_i) \neq \mathit{false}$ for all $i \in \{1, \ldots, n\}$, and from the definition of $\Omega(\varphi)$ in Table 2 we have $\Omega(\varphi) \neq \mathit{false}$. By assumption we have $\Omega(\varphi) \neq ?$, leaving only $\Omega(\varphi_1 \wedge \cdots \wedge \varphi_n) = \mathit{true}$ as the conclusion.
  - Implication (2): Assume $\Omega(\varphi) = \mathit{true}$. Then we have $\Omega(\varphi_i) = \mathit{true}$ for all $i \in \{1, \ldots, n\}$, from the definition of $\Omega(\varphi)$ in Table 2. Due to the induction hypothesis we have $M_0 \models \varphi_i$ for all i and $M_0 \models \varphi_1 \wedge \cdots \wedge \varphi_n$ follows from the semantics of $\varphi_1 \wedge \cdots \wedge \varphi_n$.
- $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$: This case is analogous to the conjunction case.
- $\varphi = AX\varphi'$: Assume that $\Omega(\varphi) \neq ?$ is true. Since we by assumption have $\Omega(\varphi) \neq ?$, and $\Omega(\varphi) = \mathit{false}$ can never occur due to the definition of $\Omega(\varphi)$ in Table 2, we must have $M_0 \models \mathit{deadlock}$ and $\Omega(AX\varphi') = \mathit{true}$. If $M_0 \models \mathit{deadlock}$ then $M_0 \models AX\varphi'$ trivially follows from the semantics of $AX\varphi'$. We therefore have $M_0 \models AX\varphi'$ iff $\Omega(AX\varphi') = \mathit{true}$ follows.
- $\varphi = EX\varphi'$: This case is analogous to the $AX\varphi'$ case.
- $\varphi = EG\varphi'$: By structural induction we have if $\Omega(\varphi') \neq ?$ then $M_0 \models \varphi'$ iff $\Omega(\varphi') = \mathit{true}$. Assume that $\Omega(\varphi) \neq ?$ is true. Since we by assumption have $\Omega(\varphi) \neq ?$, and $\Omega(\varphi) = \mathit{true}$ can never occur due to the definition of $\Omega(\varphi)$ in Table 2, we must have $\Omega(\varphi') = \mathit{false}$ and $\Omega(EG\varphi') = \mathit{false}$. Due to the induction hypothesis we have $M_0 \not\models \varphi'$, and $M_0 \not\models EG\varphi'$ follows from the semantics of $EG\varphi'$. We therefore have $M_0 \models EG\varphi'$ iff $\Omega(EG\varphi') = \mathit{true}$ follows.
- $\varphi = AG\varphi'$: This case is analogous to the $EG\varphi'$ case.
- $\varphi = EF\varphi'$: By structural induction we have if $\Omega(\varphi') \neq ?$ then $M_0 \models \varphi'$ iff $\Omega(\varphi') = \mathit{true}$. Assume that $\Omega(\varphi) \neq ?$ is true. Since we by assumption have $\Omega(\varphi) \neq ?$, and $\Omega(\varphi) = \mathit{false}$ can never occur due to the definition of $\Omega(\varphi)$ in Table 2, we must have $\Omega(\varphi') = \mathit{true}$ and $\Omega(EF\varphi') = \mathit{true}$. Due to the induction hypothesis we have $M_0 \models \varphi'$, and $M_0 \models EF\varphi'$ follows from the semantics of $EF\varphi'$. We therefore have $M_0 \models EF\varphi'$ iff $\Omega(EF\varphi') = \mathit{true}$ follows.

$\varphi = AF\varphi'$: This case is analogous to the $EF\varphi'$ case.

$\varphi = E(\varphi_1 U \varphi_2)$: For all $i \in \{1, 2\}$ by structural induction we have if $\Omega(\varphi_i) \neq ?$ then $M_0 \models \varphi_i$ iff $\Omega(\varphi_i) = \mathit{true}$. Assume that $\Omega(\varphi) \neq ?$ is true. We need to show the following two implications: (1) if $M_0 \models E(\varphi_1 U \varphi_2)$ then $\Omega(E(\varphi_1 U \varphi_2)) = \mathit{true}$, and (2) if $\Omega(E(\varphi_1 U \varphi_2)) = \mathit{true}$ then $M_0 \models E(\varphi_1 U \varphi_2)$.

- Implication (1): Assume $M_0 \models E(\varphi_1 U \varphi_2)$. Since we by assumption have $\Omega(\varphi) \neq ?$, there are two additional cases from the definition of $\Omega(\varphi)$ in Table 2: $\Omega(\varphi_2) = \mathit{true}$ or $\Omega(\varphi_1) = \Omega(\varphi_2) = \mathit{false}$. Assume for the sake of contradiction $\Omega(\varphi_1) = \Omega(\varphi_2) = \mathit{false}$ is true. Then from the induction hypothesis we have $M_0 \not\models \varphi_1$ and $M_0 \not\models \varphi_2$, implying $M_0 \not\models E(\varphi_1 U \varphi_2)$ from the semantics of $E(\varphi_1 U \varphi_2)$. This contradicts our assumption that $M_0 \models E(\varphi_1 U \varphi_2)$, and leaves us only with the first case $\Omega(\varphi_2) = \mathit{true}$. We have $\Omega(E(\varphi_1 U \varphi_2)) = \mathit{true}$ trivially follows from the definition of $\Omega(\varphi)$ in Table 2.
- Implication (2): Assume $\Omega(E(\varphi_1 U \varphi_2)) = \mathit{true}$. Then we have $\Omega(\varphi_2) = \mathit{true}$ from the definition of $\Omega(\varphi)$ in Table 2. Due to the induction hypothesis we have $M_0 \models \varphi_2$, and $M_0 \models E(\varphi_1 U \varphi_2)$ follows from the semantics of $E(\varphi_1 U \varphi_2)$.

$\varphi = A(\varphi_1 U \varphi_2)$: This case is analogous to the $E(\varphi_1 U \varphi_2)$ case.

□

B Proof of Theorem 2

Proof. The proof is by structural induction on φ. As a sketch, we will here prove the correctness of the rules for $EF\varphi$ and $E\varphi_1 U \varphi_2$. The proofs for the other rules are analogous.

$\varphi = EF\varphi'$: By structural induction we have $M \models \varphi'$ iff $M \models \rho(\varphi')$. We need to show the following two implications: (1) if $M \models EF\varphi'$ then $M \models \rho(EF\varphi')$, and (2) if $M \models \rho(EF\varphi')$ then $M \models EF\varphi'$.

- Implication (1): Assume $M \models EF\varphi'$. Then there exists $M' \in \mathcal{M}(N)$ s.t. $M \to^* M'$ and $M' \models \varphi'$. Due to the induction hypothesis we have $M' \models \rho(\varphi')$. There are now 6 cases given by the definition of $\rho(EF\varphi')$ in Table 3. The otherwise case is trivial due to the induction hypothesis.
  - Case $\rho(\varphi') = \neg \mathit{deadlock}$: If $M' \models \neg \mathit{deadlock}$ then we must also have $M \models \neg \mathit{deadlock}$, as $M \to^* M'$ and $en(M) \neq \emptyset$.
  - Case $\rho(\varphi') = EF\varphi''$: There exists $M'' \in \mathcal{M}(N)$ s.t. $M' \to^* M''$ and $M'' \models \varphi''$. Since we have $M \to^* M'$ and $M' \to^* M''$ we must also have $M \to^* M''$ is the case, implying that $M \models EF\varphi''$ is true.
  - Case $\rho(\varphi') = AF\varphi''$: Clearly there exists $M'' \in \mathcal{M}(N)$ s.t. $M' \to^* M''$ and $M'' \models \varphi''$ by the semantics of $AF\varphi''$. Since we have $M \to^* M'$ and $M' \to^* M''$ we must also have $M \to^* M''$ is the case, implying that $M \models EF\varphi''$ is true.
  - Case $\rho(\varphi') = E(\varphi_1 U \varphi_2)$: Clearly there exists $M'' \in \mathcal{M}(N)$ s.t. $M' \to^* M''$ and $M'' \models \varphi_2$ by the semantics of $E(\varphi_1 U \varphi_2)$. Since we have $M \to^* M'$ and $M' \to^* M''$ we must also have $M \to^* M''$ is the case, implying that $M \models EF\varphi_2$ is true.

- Case $\rho(\varphi') = A(\varphi_1 U \varphi_2)$: Clearly there exists $M'' \in \mathcal{M}(N)$ s.t. $M' \rightarrow^* M''$ and $M'' \models \varphi_2$ by the semantics of $A(\varphi_1 U \varphi_2)$. Since we have $M \rightarrow^* M'$ and $M' \rightarrow^* M''$ we must also have $M \rightarrow^* M''$ is the case, implying that $M \models EF\varphi_2$ is true.
- Case $\rho(\varphi') = \varphi_1 \vee \cdots \vee \varphi_n$: Due to the semantics of $\varphi_1 \vee \cdots \vee \varphi_n$ there exists i s.t. $1 \leq i \leq n$ and $M' \models \varphi_i$. From $M' \models \varphi_i$ we have $M \models EF\varphi_i$, and $M \models EF\varphi_1 \vee \cdots \vee EF\varphi_n$ follows trivially from disjunction introduction.

– Implication (2): Assume $M \models \rho(EF\varphi')$. There are 6 cases given by the definition of $\rho(EF\varphi')$ in Table 3. The otherwise case is trivial due to the induction hypothesis.
- Case $\rho(\varphi') = \neg deadlock$: Trivially we have that $M \models \neg deadlock$ implies $M \models EF\neg deadlock$ by the semantics of φ.
- Case $\rho(\varphi') = EF\varphi''$: Trivially we have that $M \models EF\varphi''$ implies $M \models EFEF\varphi''$ by the semantics of φ.
- Case $\rho(\varphi') = AF\varphi''$: By the induction hypothesis if $M \models \rho(AF\varphi'')$ then we have $M \models AF\varphi''$. Trivially we have that $M \models AF\varphi''$ implies $M \models EFAF\varphi''$ by the semantics of φ.
- Case $\rho(\varphi') = E(\varphi_1 U \varphi_2)$: By the induction hypothesis if $M \models \rho(E(\varphi_1 U \varphi_2))$ then we have $M \models E(\varphi_1 U \varphi_2)$. Trivially we have that $M \models E(\varphi_1 U \varphi_2)$ implies $M \models EFE(\varphi_1 U \varphi_2)$ by the semantics of φ.
- Case $\rho(\varphi') = A(\varphi_1 U \varphi_2)$: By the induction hypothesis if $M \models \rho(A(\varphi_1 U \varphi_2))$ then we have $M \models A(\varphi_1 U \varphi_2)$. Trivially we have that $M \models A(\varphi_1 U \varphi_2)$ implies $M \models EFA(\varphi_1 U \varphi_2)$ by the semantics of φ.
- Case $\rho(\varphi') = \varphi_1 \vee \cdots \vee \varphi_n$: By the induction hypothesis if $M \models \rho(EF\varphi_1 \vee \cdots \vee EF\varphi_n)$ then we have $M \models EF\varphi_1 \vee \cdots \vee EF\varphi_n$. Due to the semantics of $EF\varphi_1 \vee \cdots \vee EF\varphi_n$ there exists i s.t. $1 \leq i \leq n$ and $M \models EF\varphi_i$. There exists $M' \in \mathcal{M}(N)$ s.t. $M \rightarrow^* M'$ and $M' \models \varphi_i$. By disjunction introduction we have $M' \models \varphi_1 \vee \cdots \vee \varphi_n$, and $M \models EF(\varphi_1 \vee \cdots \vee \varphi_n)$ follows since $M \rightarrow^* M'$.

$\varphi = E(\varphi_1 U \varphi_2)$: By structural induction we have $M \models \varphi_1$ iff $M \models \rho(\varphi_1)$ and $M \models \varphi_2$ iff $M \models \rho(\varphi_2)$. We need to show the following two implications: (1) if $M \models E(\varphi_1 U \varphi_2)$ then $M \models \rho(E(\varphi_1 U \varphi_2))$, and (2) if $M \models \rho(E(\varphi_1 U \varphi_2))$ then $M \models E(\varphi_1 U \varphi_2)$.

– Implication (1): Assume $M \models E(\varphi_1 U \varphi_2)$. Then there exists $\pi \in \Pi^{max}(M)$ and a position i s.t. $\pi_i \models \varphi_2$ and for all $j \in \{0, \ldots, i-1\}$ we have $\pi_j \models \varphi_1$. There are 5 cases given by the definition of $\rho(E(\varphi_1 U \varphi_2))$ in Table 3. The otherwise case is trivial due to the induction hypothesis.
- Case $\rho(\varphi_2) = \neg deadlock$: If $\pi_i \models \neg deadlock$ then we must also have $M \models \neg deadlock$, as $M \rightarrow^* \pi_i$ and $en(\pi_i) \neq \emptyset$.
- Case $\rho(\varphi_1) = deadlock$: Then the only case where $M \models E(\varphi_1 U \varphi_2)$ can be true is when $i = 0$, implying $M \models \varphi_2$. By the induction hypothesis we conclude with $M \models \rho(\varphi_2)$.
- Case $\rho(\varphi_1) = \neg deadlock$: Clearly, for any path we have $\neg deadlock$ always holds in every intermidiary marking due to the definition of

paths, giving us $M \models E(\mathit{true}\, U \varphi_2)$. This is the definition of $M \models EF\varphi_2$ for the minimal set of CTL operators.
  - Case $\rho(\varphi_2) = EF\varphi_3$: There exists $M' \in \mathcal{M}(N)$ s.t. $\pi_i \to^* M'$ and $M' \models \varphi_3$. Since we have $M \to^* \pi_i$ and $\pi_i \to^* M'$ we must also have $M \to^* M'$ is the case, implying that $M \models EF\varphi_3$ is true.
  - Case $\rho(\varphi_2) = \varphi_3 \vee EF\varphi_4$: Either we have $\pi_i \models \varphi_3$ or there exists $M' \in \mathcal{M}(N)$ s.t. $\pi_i \to^* M'$ and $M' \models EF\varphi_4$. In the former case clearly we have $M \models E(\varphi_1 U \varphi_3)$ since the path π exists, and we can conlude with $M \models (EF\varphi_4) \vee E(\varphi_1 U \varphi_3)$ by disjunction introduction. In the latter case since $M \to^* \pi_i$ and $\pi_i \to^* M'$ we must also have $M \to^* M'$ is the case, implying that $M \models EF\varphi_4$ is true, and we can conclude with $M \models (EF\varphi_4) \vee E(\varphi_1 U \varphi_3)$ by disjunction introduction.

– Implication (2): Assume $M \models \rho(E(\varphi_1 U \varphi_2))$. There are 5 cases given by the definition of $\rho(E(\varphi_1 U \varphi_2))$ in Table 3. The otherwise case is trivial due to the induction hypothesis.
  - Case $\rho(\varphi_2) = \neg \mathit{deadlock}$: Trivially we have that $M \models \neg \mathit{deadlock}$ implies $M \models E(\varphi_1 U \neg \mathit{deadlock})$ by the semantics of φ.
  - Case $\rho(\varphi_1) = \mathit{deadlock}$: Trivially we have that $M \models \varphi_2$ implies $M \models E(\varphi_1 U \varphi_2)$ by the semantics of φ.
  - Case $\rho(\varphi_1) = \neg \mathit{deadlock}$: Trivially we have that $M \models EF\varphi_2$ implies $M \models E(\neg \mathit{deadlock}\, U \varphi_2)$ by the semantics of φ.
  - Case $\rho(\varphi_2) = EF\varphi_3$: Trivially we have that $M \models EF\varphi_3$ implies $M \models E(\neg \mathit{deadlock}\, U EF\varphi_3)$ by the semantics of φ.
  - Case $\rho(\varphi_2) = \varphi_3 \vee EF\varphi_4$: If $M \models (EF\varphi_4) \vee E(\varphi_1 U \varphi_3)$ then either we have $M \models EF\varphi_4$ or $M \models E(\varphi_1 U \varphi_3)$. In the former case by disjunction introduction we have $M \models \varphi_3 \vee EF\varphi_4$, and trivially we have $M \models E(\varphi_1 U \varphi_3 \vee EF\varphi_4)$ by the semantics of φ. In the later case there exists $\pi \in \Pi^{max}(M)$ and a position i s.t. $\pi_i \models \varphi_3$ and for all $j \in \{0, \ldots, i-1\}$ we have $\pi_j \models \varphi_1$. By disjunction introduction we have $\pi_i \models M \models \varphi_3 \vee EF\varphi_4$, and clearly since the path π exists we have $M \models E(\varphi_1 U \varphi_3 \vee EF\varphi_4)$.

□

C Proof of Theorem 3

Proof. We prove the three claims by structural induction on φ.
Base Cases:

- $\varphi = \mathit{true}$: Since $\mathit{simplify}(\mathit{true}) = (\mathit{true}, \{\{0 \leq 1\}\}, \emptyset)$ the formula remains unchanged and Condition 1 trivially holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution and Condition 3 is a vacuous case.
- $\varphi = \mathit{false}$: Since $\mathit{simplify}(\mathit{false}) = (\mathit{false}, \emptyset, \{\{0 \leq 1\}\})$ the formula remains unchanged and Condition 1 trivially holds. Condition 2 is a vacuous case and Condition 3 holds as any variable assignment is a solution to $\{0 \leq 1\}$.

$\varphi = \mathit{deadlock}$: Since $\mathit{simplify}(\mathit{deadlock}) = (\mathit{deadlock}, \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$ the formula remains unchanged and Condition 1 trivially holds. Conditions 2 and 3 hold as any variable assignment is a solution to $\{0 \leq 1\}$.

$\varphi = e_1 \bowtie e_2$: If either $\mathit{const}(e_1)$ or $\mathit{const}(e_2)$ is not linear, then $\mathit{simplify}(e_1 \bowtie e_2) = (e_1 \bowtie e_2, \{\{0 \leq 1\}\}, \{\{0 \leq 1\}\})$ and all three conditions trivially hold. Otherwise, we have $\mathit{LPS} = \{\{\mathit{const}(e_1) \bowtie \mathit{const}(e_2)\}\}$ and $\overline{\mathit{LPS}} = \{\{\mathit{const}(e_1) \mathbin{\overline{\bowtie}} \mathit{const}(e_2)\}\}$. Let M be a marking such that $M_0 \xrightarrow{w} M$. We will now argue that the three conditions of the theorem hold. There are three subcases to consider:

- Algorithm 1 returns $\mathit{simplify}(\mathit{false}) = (\mathit{false}, \emptyset, \{\{0 \leq 1\}\})$ because $\{\mathit{LP} \cup \mathit{BASE} \mid \mathit{LP} \in \mathit{LPS}\}$ has no solution. Then $M \not\models e_1 \bowtie e_2$ as otherwise $\wp(w)$ would be a solution both to BASE as well as $\{\mathit{const}(e_1) \bowtie \mathit{const}(e_2)\}$ due to the construction of the state equations for e_1 and e_2. This means that Condition 1 holds, Condition 2 is vacuous, and Condition 3 holds as $\wp(w)$ is clearly a solution to $\{0 \leq 1\}$.
- Algorithm 1 returns $\mathit{simplify}(\mathit{true}) = (\mathit{true}, \{\{0 \leq 1\}\}, \emptyset)$ because $\{\mathit{LP} \cup \mathit{BASE} \mid \mathit{LP} \in \overline{\mathit{LPS}}\}$ has no solution. Then $M \not\models e_1 \mathbin{\overline{\bowtie}} e_2$ as otherwise $\wp(w)$ would be a solution both to BASE as well as $\{\mathit{const}(e_1) \mathbin{\overline{\bowtie}} \mathit{const}(e_2)\}$ due to the construction of the state equations for e_1 and e_2. This implies that $M \models e_1 \mathbin{\overline{\bowtie}} e_2$ and hence Condition 1 holds. Condition 2 holds as $\wp(w)$ is clearly a solution to $\{0 \leq 1\}$ and Condition 3 is vacuous.
- Algorithm 1 returns $(e_1 \bowtie e_2, \mathit{LPS}, \overline{\mathit{LPS}})$ and because the formula was unchanged, Condition 1 trivially holds. Due to the construction of the linear programs based on state equations, it is clear that if $M \models e_1 \bowtie e_2$ then $\wp(w)$ is a solution to both BASE and LPS, implying Condition 2. Symmetrically, if $M \not\models e_1 \bowtie e_2$ then $M \models e_1 \mathbin{\overline{\bowtie}} e_2$ and hence $\wp(w)$ is a solution to both BASE and $\overline{\mathit{LPS}}$, meaning that Condition 3 holds too.

Inductive Cases (where $M_0 \xrightarrow{w} M$):

$\varphi = \neg\varphi_1$: Let $\mathit{simplify}(\varphi_1) = (\varphi_1', \mathit{LPS}, \overline{\mathit{LPS}})$. By structural induction hypothesis we know that $M \models \varphi_1$ iff $M \models \varphi_1'$ which implies that $M \models \neg\varphi_1$ iff $M \models \neg\varphi_1'$ and Condition 1 is thus satisfied for all three possible returns. Conditions 2 and 3 clearly hold if $\mathit{simplify}(\mathit{false})$ or $\mathit{simplify}(\mathit{true})$ is returned. In case the return value is $(\neg\varphi_1', \overline{\mathit{LPS}}, \mathit{LPS})$, we use the induction assumption that Conditions 2 and 3 hold for φ_1 and by adding the negation to φ_1 and swapping the sets of linear programs, the conditions hold also for $\neg\varphi_1$.

$\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$: Let $\mathit{simplify}(\varphi_i) = (\varphi_i', \mathit{LPS}_i, \overline{\mathit{LPS}_i})$ for all i, $1 \leq i \leq n$. By structural induction hypothesis, for all i we have $M \models \varphi_i$ iff $M \models \varphi_i'$. Hence if for some i it is the case that $\varphi_i' = \mathit{false}$ we can terminate and return $\mathit{simplify}(\mathit{false})$ since it is clear $M \not\models \varphi$ and all three conditions hold as required. Similarly, if $\varphi_i' = \mathit{true}$ for some i, then this conjunct can be safely skipped over as it will not change the validity of φ'. Moreover, should this be the case for all i, $1 \leq i \leq n$, then we can safely return $\mathit{simplify}(\mathit{true})$ and all three conditions still hold.

Assume that $M \models \varphi$, then clearly $M \models \varphi_i$ for all i and by the induction hypothesis $\wp(w)$ is a solution to each LPS_i, meaning that for each i there is an $LP_i \in LPS_i$ for which $\wp(w)$ is a solution. By the definition of the merge operation, we know that there exists an LP such that $LP \subseteq LP_1 \cup LP_2 \cup \ldots \cup LP_n$, where $LP \in LPS$ and $\wp(w)$ is a solution to LP. As a consequence, LPS has $\wp(w)$ as a solution and Condition 2 is thus satisfied. Conversely, if LPS has no solution, this implies $M \not\models \varphi$ and in this case we can safely return *simplify*(*false*).

Let us assume that $M \not\models \varphi$, implying that $M \not\models \varphi_i$ for at least one i. By induction hypothesis, there is $LP \in \overline{LPS}_i$ such that $\wp(w)$ is a solution to LP and because we perform unions of all $\overline{LPS}_i$, clearly $LP \in \overline{LPS}$ and Condition 3 therefore holds.

$\varphi = \varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$: Let $\mathit{simplify}(\varphi_i) = (\varphi_i', LPS_i, \overline{LPS}_i)$ for all i, $1 \leq i \leq n$. By structural induction hypothesis, for all i we have $M \models \varphi_i$ iff $M \models \varphi_i'$. Hence if for some i it is the case that $\varphi_i' = \mathit{true}$ we can terminate and return *simplify*(*true*) since it is clear $M \models \varphi$ and all three conditions hold as required. Similarly, if $\varphi_i' = \mathit{false}$ for some i, then this conjunct can be safely skipped over as it will not change the validity of φ'. Moreover, should this be the case for all i, $1 \leq i \leq n$, then we can safely return *simplify*(*false*) and all three conditions still hold.

Assume that $M \models \varphi$, then clearly $M \models \varphi_i$ for some i and by the induction hypothesis $\wp(w)$ is a solution to some $LP \in LPS_i$. Then the algorithm either returns *simplify*(*true*) and all three conditions hold, or $LP \in LPS$ as we perform the union operation on LPS and this guarantees that Condition 2 holds once the algorithm returns $(\varphi', LPS, \overline{LPS})$.

Let us assume that $M \not\models \varphi$, implying that $M \not\models \varphi_i$ for all i. By induction hypothesis, for all i there is $LP_i \in \overline{LPS}_i$ such that $\wp(w)$ is a solution to LP. By the definition of the merge operation, we know that there exists an LP such that $LP \subseteq LP_1 \cup LP_2 \cup \ldots \cup LP_n$, where $LP \in \overline{LPS}$ and $\wp(w)$ is a solution to LP. As a consequence, $\overline{LPS}$ has $\wp(w)$ as a solution and Condition 3 is thus satisfied. Conversely, if $\overline{LPS}$ has no solution, this implies $M \models \varphi$ and in this case we can safely return *simplify*(*true*).

$\varphi = QX\varphi_1$, where $Q \in \{A, E\}$: Let $\mathit{simplify}(\varphi_1) = (\varphi_1', LPS, \overline{LPS})$. By structural induction hypothesis, we have $M \models \varphi_1$ iff $M \models \varphi_1'$. In case that φ_1' is either *true* or *false*, the four cases in the algorithm clearly preserve logical equivalence and all three conditions are satisfied. Otherwise we return $QX\varphi_1'$ which is equivalent to $QX\varphi_1$ and Condition 1 remaind satisfied. Both sets of linear programs that are returned have any assignment as a solution, so Conditions 2 and 3 hold too.

$\varphi = QP\varphi_1$ where $QP \in \{AG, EG, AF, EF\}$: This case is analogous to the next operators discussed above.

$\varphi = Q(\varphi_1 U \varphi_2)$ where $Q \in \{A, E\}$: Let $\mathit{simplify}(\varphi_1) = (\varphi_1', LPS_1, \overline{LPS}_1)$ and $\mathit{simplify}(\varphi_2) = (\varphi_2', LPS_2, \overline{LPS}_2)$. By structural induction hypothesis, we have $M \models \varphi_1$ iff $M \models \varphi_1'$, and $M \models \varphi_2$ iff $M \models \varphi_2'$. If $\varphi_2' = \mathit{true}$ then $Q(\varphi_1 U \varphi_2)$ is equivalent to *true* and we can return *simplify*(*true*) while satisfying all three conditions. Similarly if $\varphi_2' = \mathit{false}$ we can safely re-

turn *simplify*(*false*). If $\varphi_1' = \mathit{true}$ then $Q(\varphi_1 U \varphi_2)$ becomes logically equivalent to $QF\varphi_2'$ and both sets of linear programs that are returned have any assignment as a solution, so all three conditions are satisfied. In case $\varphi_1' = \mathit{false}$ then necessarily φ_2 must hold immediately and we can return $(\varphi_2', LPS_2, \overline{LPS}_2)$ that satisfies all three conditions by the induction hypothesis. Otherwise we return $Q(\varphi_1' U \varphi_2')$ that is equivalent to φ and the two returned linear programs admit all assignments as solutions, so all three conditions hold.

□