Summary

State space analysis of distributed systems is often a time and resource consuming task. It involves checking whether a system satisfies a specification, and is traditionally called *model checking*. Systems are often modelled using high-level modelling formalisms, an example being the Petri net formalism that also offers a graphical modelling notation. Specifications can be expressed using, for example, *computation tree logic*, and often contain safety requirements, that guarantee the absence of unwanted behaviour in the system.

Explicit analysis methods perform an exhaustive search in the state space, generated by considering every interleaving action in the analysed system. The number of states can be of exponential size and is commonly referred to as the state-space explosion problem. This makes model checking of real-world systems require an infeasible amount of time and resources.

There exist many tools to perform model checking, which all try to circumvent the state explosion problem, by using different verification techniques. TAPAAL is an open-source tool facilitating both a graphical user interface for system modelling of Petri nets, and a variety of verification engines, each optimised to cope with different verification challenges. One of these techniques is the structural reduction technique, which removes redundant places and transitions in Petri nets. Another technique is state equations, which uses linear overapproximation to determine the satisfiability of predicates in the initial marking. An efficient state space representation is used, which improves successor generation, and reduces the required memory for storing states. Other techniques for explicitly reducing the size of the state space include partial order reduction and symmetry reduction techniques, both of which are implemented in the Low Level Petri net Analyzer (LoLA).

We present our interpretation of the well-known partial order reduction technique called *stubborn sets* for Petri net. We name this interpretation *stubborn reduction* and define it on the more abstract level of Labelled Transition Systems (LTS).

We define how stubborn reduction guarantee the preservation of reachability properties and prove its correctness. We apply this to Petri nets with inhibitor arcs. This is done by generating a subset of transitions named the *interesting set* of transitions, which are the only transitions that can affect the evaluation of a given a formula. The interesting set of transitions is then transformed into a stubborn set by a closure algorithm, for which we prove its termination and correctness.

We present an interpretation of deciding the siphon-trap property for Petri net, which, instead of characterising the problem of deciding the siphon-trap property as a Boolean satisfiability problem, is characterised as an integer linear program. If a Petri net exhibits the siphon-trap property then the Petri net is deadlock free. Furthermore, we parameterise the procedure with a max depth, bounding the number of decision variables and constraints.

The size of the interesting set of transitions is based on the size of the formula in question. If the formula is large, all transitions can potentially be added to

the set and subsequently include all transitions in the stubborn set, leaving the stubborn reduction effectless. For this reason, we introduce the *formula simplification* technique, which is an extension of the existing state equations for Petri net used in TAPAAL. We traverse the structure of a given formula and identify subformulae that are either trivially satisfied or impossible to satisfy, and replace them with easier to verify alternatives. These formulae are identified using integer linear programming such that the generated integer linear program has a solution if the formula or subformula is satisfied. We provide an algorithm for performing the simplification and prove its correctness. Additionally, we further extend the logic to include CTL operators. Furthermore, formula simplification can also be used for performing sanity checks on specifications. If the formula or parts of it can be simplified to trivially true or false, then this may indicate a faulty or uninteresting specification.

The feasibility of model checking is strongly dependent on the efficiency of the verification technique. Our experiments show that many real-world systems generates an unmanageable amount of states, and that certain countermeasures have to be taken, in order to provide meaningful information. We have also seen, that many formulae contains trivial subformulae, that can be answered in the initial state, possibly because they were automatically generated.

We examined the combined performance of every implemented technique in TAPAAL and compared it with LoLA. On all categories combined, TAPAAL provided more exclusive answers and used less memory. LoLA performed faster verification and generally explored fewer states. Specifically, LoLA performed better on reachability formulae and TAPAAL performed better on CTL formulae.

A Simplified and Stubborn Approach to CTL Model Checking of Petri Nets

Frederik Bønneland, Jakob Dyhr, and Mads Johannsen

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.
`{fbanne12,jdyhr12,mjohan12}@student.aau.dk`

Abstract. We present an interpretation of stubborn sets, described using labelled transitions systems. The approach is applied in the context of Petri nets with inhibitor arcs. We extend reachability preserving stubborns sets to include preservation of fireability, cardinality, and deadlock properties, and base it on generating an interesting set of transitions for a given formula. We introduce an interpretation for deciding the siphon-trap property for Petri nets using integer linear programming. Based on the theory of state equations, we construct a CTL formula simplification technique that removes trivially verifiable subformulae. We implement and benchmark the individual and combined techniques against the state of the art verification tools in TAPAAL and LoLA. The integer linear programming interpretation of the siphon-trap property shows potential in analysis of deadlock freedom, but the implementation performs worse than the Boolean satisfiability representation implemented in LoLA. Our experiments show that the combination of structural reduction and stubborn reduction generally yields considerable performance improvements, but that the combination conflicts on some models. Formula simplification is able to reduce a significant number of formulae to questions that are trivially answered, and the simplified formulae improve the stubborn and structural reduction techniques. Issues with simplification of fireability formulae were discovered. On all formula categories combined, we provide more exclusive answers and consume less memory while LoLA perform faster verification and generally explore fewer states. All mentioned techniques have been implemented in the open-source Petri net verification tool TAPAAL.

1 Introduction

State space analysis of distributed systems is often a time and resource consuming task. It involves checking whether a system satisfies a specification, and is traditionally called *model checking* [4]. Systems are often modelled using high-level modelling formalisms, an example being the Petri net [19] formalism that also offers a graphical modelling notation. Specifications can be expressed using, for example, *computation tree logic*, and often contain safety requirements, that guarantee the absence of unwanted behaviour in the system.

Explicit analysis methods perform an exhaustive search in the state space, generated by considering every interleaving action in the analysed system. The number of states can be of exponential size and is commonly referred to as the state-space explosion problem. This makes model checking of real-world systems require an infeasible amount of time and resources.

There exist many tools to perform model checking, which all try to circumvent the state explosion problem, by using different verification techniques. TAPAAL [11,6] is an open-source tool facilitating both a graphical user interface for system modelling of Petri nets, and a variety of verification engines, each optimised to cope with different verification challenges. One of these techniques is the structural reduction technique, which removes redundant places and transitions in Petri nets [11]. Another technique is state equations, which uses linear overapproximation to determine the satisfiability of predicates in the initial marking. An efficient state space representation is used [13], which improves successor generation, and reduces the required memory for storing states. Other techniques for explicitly reducing the size of the state space include partial order reduction [26] and symmetry reduction [21] techniques, both of which are implemented in the Low Level Petri net Analyzer (LoLA) [28]. Alternatively to explicit analysis, it is possible to perform symbolic analysis, such as the Interval Decision Diagrams [22] used in the Marcie verification tool [10].

We present our interpretation of the well-known partial order reduction technique called *stubborn sets* [24,20,26] for Petri net. We name this interpretation *stubborn reduction* and define it on the more abstract level of Labelled Transition Systems (LTS).

We define how stubborn reduction guarantee the preservation of reachability properties and prove its correctness. We apply this to Petri nets with inhibitor arcs. This is done by generating a subset of transitions named the *interesting set* of transitions, which are the only transitions that can affect the evaluation of a given a formula. The interesting set of transitions is then transformed into a stubborn set by a closure algorithm, for which we prove its termination and correctness.

We present an interpretation of deciding the siphon-trap property [9] for Petri net, which, instead of characterising the problem of deciding the siphon-trap property as a Boolean satisfiability problem [18], is characterised as an integer linear program. If a Petri net exhibits the siphon-trap property then the Petri net is deadlock free. Furthermore, we parameterise the procedure with a max depth, bounding the number of decision variables and constraints.

The size of the interesting set of transitions is based on the size of the formula in question. If the formula is large, all transitions can potentially be added to the set and subsequently include all transitions in the stubborn set, leaving the stubborn reduction effectless. For this reason, we introduce the *formula simplification* technique, which is an extension of the existing state equations for Petri net used in TAPAAL. We traverse the structure of a given formula and identify subformulae that are either trivially satisfied or impossible to satisfy, and replace them with easier to verify alternatives. These formulae are identified using integer linear programming such that the generated integer linear program has a solution if the formula or subformula is satisfied. We provide an algorithm for performing the simplification and prove its correctness. Additionally, we further extend the logic to include CTL operators. Furthermore, formula simplification can also be used for performing sanity checks on specifications. If the formula or parts of it can be simplified to trivially true or false, then this may indicate a faulty or uninteresting specification.

Related work. Parts of this thesis are based on our pre-specialisation project. Details can be found in the bibliographical remarks in Section 10. The stubborn reduction techniques is related to the work on stubborn sets [26,20,15]. Several techniques for improving the feasibility of model checking have been implemented in the LoLA tool. This includes stubborn sets [20], symmetry reduction [21], and the Counter Example Guided Abstraction Refinement (CEGAR) technique to reachability analysis of Petri nets [27]. Weak semi stubborn reduction and reachability preserving stubborn reduction rules together with proofs are drawn from [20,26,15]. We contribute by lifting the mentioned stubborn set rules to a more abstract interpretation and provide a reachability preserving stubborn reduction rule. The reachability preserving closure for Petri nets is based on [15,26], and is extended to include inhibitor arcs. The algorithm for performing the stubborn set closure applies the method first described in [25], and we prove its correctness. The integer linear programming interpretation of the procedure for deciding the siphon-trap property is based on the work in [18]. We use attractor set theory in [20] and contribute with extending it to a larger reachability logic to provide a formal and syntactically defined set of transitions that form the base of reachability preserving stubborn sets. In [11] the authors used the integer linear programming technique state equations for Petri net to verify cardinality queries, which we extend to a larger CTL logic and base our formula simplification procedure upon. In [27] the authors also used the state equations technique and is the base for deciding reachability using CEGAR. We contribute by demonstrating the performance on a database of models, consisting of both industrial and academic models and queries, from the 2017 Model Checking Contest (MCC) [2] when combining reduction techniques.

Acknowledgements. We thank Torsten Liebke from Rostock University, who provided insight into the techniques used by the model checker LoLA. We thank Peter Gjøl Jensen for his technical assistance with our implementation. We would also like to show our gratitude to our supervisor Jiří Srba for providing stimulating feedback throughout our Master's thesis.

In Section 2 we introduce preliminary definitions of LTS, Petri net, CTL and reachability logic, and integer linear programming. In Section 3 we present stubborn reduction using LTS and prove the preservation of reachability properties. In Section 4 we apply stubborn reduction to Petri net with inhibitor arcs, which includes the theory of interesting set of transitions, proving that the application is correct, and providing an algorithm for computing the stubborn set closure, including termination and correctness proofs. In Section 5 we present our integer linear programming interpretation for deciding the siphon-trap property. In Section 6 we present the formula simplification procedure and prove its correctness. In Section 7 we give an overview of the TAPAAL toolchain, and highlight some of the interesting parts of the implementation done. In Section 8 we present our experimental setup and results on the database of known models from MCC'17. In Section 9 we conclude our findings.

2 Preliminaries

A Labelled Transition System (LTS) is a tuple $(\mathcal{S}, A, \rightarrow)$ 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. Whenever $(s, a, s') \in \rightarrow$, we write $s \xrightarrow{a} s'$ and say that a is enabled in s, and we can *execute* a in s yielding s'. Otherwise we say that a is disabled in s and write $s \overset{a}{\not\rightarrow}$. The set of all enabled actions in a state s is denoted $en(s)$. A state s is said to be a *deadlock* if $en(s) = \emptyset$. For a possibly infinite sequence of actions $w = a_1 a_2 \cdots \in A^* \cup A^\omega$ and states $s_1, s_2, \ldots$ we call w an *action sequence* if $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$. If w is finite then this is written as $s_1 \xrightarrow{w} s_n$. By convention $s \xrightarrow{\varepsilon} s$ always holds, where ε is the empty action sequence. Any action sequence of length n from s to s' is written as $s \rightarrow^n s'$. If there exists an action sequence $w \in A^*$ such that $s \xrightarrow{w} s'$, we write $s \rightarrow^* s'$. The set of all reachable states from a state s is given by the set $reach(s) = \{s' \mid s \rightarrow^* s'\}$. The sequence of states induced by an action sequence is called a *path* and is written as $\pi = s_1 s_2 \cdots$. We use $\Pi(s)$ to denote the set of all paths starting from a state s, and $\Pi = \bigcup_{s \in \mathcal{S}} \Pi(s)$ is the set of all paths. The length of a path is given by the function $\ell : \Pi \rightarrow \mathbb{N} \cup \{\infty\}$. A position i in a path $\pi \in \Pi$ refers to state s_i in the path and is written as π_i. If π is infinite then $i \in \mathbb{N}$, otherwise $1 \leq i \leq \ell(\pi)$. We use $\Pi^{max}(s)$ to denote the set of all maximal paths starting from a state s which is defined as $\Pi^{max}(s) = \{\pi \in \Pi(s) \mid \ell(\pi) = \infty \text{ or } \pi_{\ell(\pi)} \text{ is a deadlock}\}$.

2.1 Computation Tree Logic

We describe the syntax and semantics of a Computation Tree Logic (CTL) [5]. Let AP be a set of atomic propositions, $a \in AP$ an atomic proposition, and $(\mathcal{S}, A, \rightarrow)$ an LTS. We evaluate atomic propositions using the function $v : \mathcal{S} \rightarrow 2^{AP}$, where $v(s)$ is the set of atomic propositions satisfied in the state $s \in \mathcal{S}$. The CTL syntax and semantics are given as follows:

$$\begin{aligned}\varphi ::=\ & \mathit{true} \mid \mathit{false} \mid a \mid \mathit{deadlock} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \implies \varphi_2 \mid \varphi_1 \iff \varphi_2 \\ & \mid AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A(\varphi_1 U \varphi_2) \mid E(\varphi_1 U \varphi_2)\end{aligned}$$

The semantics of formula φ is defined for a state $s \in \mathcal{S}$ as follows:

$s \models true$	
$s \not\models false$	
$s \models a$	iff $a \in v(s)$
$s \models deadlock$	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 $s \not\models \varphi$
$s \models \varphi_1 \implies \varphi_2$	iff $s \not\models \varphi_1$ or $s \models \varphi_2$
$s \models \varphi_1 \iff \varphi_2$	iff ($s \models \varphi_1$ iff $s \models \varphi_2$)
$s \models AX\varphi$	iff for all $s' \in \mathcal{S}$ if $s \to s'$ then $s' \models \varphi$
$s \models EX\varphi$	iff exists $s' \in \mathcal{S}$ s.t $s \to s'$ and $s' \models \varphi$
$s \models AG\varphi$	iff for all $\pi \in \Pi^{max}(s)$ and for all positions i in π we have $\pi_i \models \varphi$
$s \models EF\varphi$	iff exists $\pi \in \Pi^{max}(s)$ s.t. there exists a position i in π s.t. $\pi_i \models \varphi$
$s \models AF\varphi$	iff for all $\pi \in \Pi^{max}(s)$ there exists a position i in π s.t. $\pi_i \models \varphi$
$s \models EG\varphi$	iff exists $\pi \in \Pi^{max}(s)$ 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 exists a position i in π s.t. $\pi_i \models \varphi_2$ and for all $1 \leq j < i$ we have $\pi_j \models \varphi_1$
$s \models E(\varphi_1 U \varphi_2)$	iff exists $\pi \in \Pi^{max}(s)$ and there exists a position i in π s.t. $\pi_i \models \varphi_2$ and for all $1 \leq j < i$ we have $\pi_j \models \varphi_1$

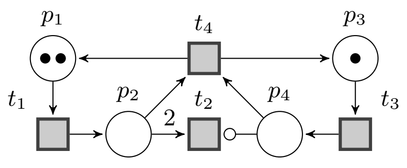
We use Φ_{CTL} to denote the set of all CTL formulae.

2.2 Petri Nets

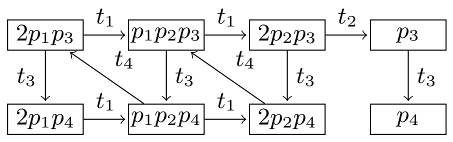
Let $\mathbb{N}^0 = \mathbb{N} \cup \{0\}$ be the set of natural numbers including 0. Let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ be the set of natural numbers including infinity.

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 where $P \cup T \neq \emptyset$, W: $(P \times T) \cup (T \times P) \to \mathbb{N}^0$ is a weight function for regular arcs, and $I : (P \times T) \to \mathbb{N}^\infty$ is a 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 place p. All markings of a Petri net N are denoted $\mathcal{M}(N)$. The initial marking of a Petri net N is denoted as $M_0 \in \mathcal{M}(N)$. For a place or transition x, we denote the *pre-set* of x as $\bullet x = \{y \mid W((y, x)) > 0\}$, and the *post-set* of x as $x\bullet = \{y \mid W((x, y)) > 0\}$. For a transition t, we denote the *inhibitor pre-set* of t as $\circ t = \{p \mid I((p, t)) < \infty\}$. For a place p, we denote the *inhibitor post-set* of p as $p\circ = \{t \mid I((p, t)) < \infty\}$. For either a set of places or transitions X we define the pre-set of X as $\bullet X = \bigcup_{x \in X} \bullet x$, and the post-set of X as $X\bullet = \bigcup_{x \in X} x\bullet$.



(a) A Petri net illustrating places, transitions, tokens, and weights.



(b) LTS describing the Petri net in Fig. 1a.

Fig. 1

The semantics of a Petri net can be described using an LTS. Given $N = (P, T, W, I)$ we define $T(N) = (\mathcal{S}, A, \rightarrow)$ as follows: $\mathcal{S} = \mathcal{M}(N)$ is the set of all markings, $A = T$ is the set of labels derived from each transition, and $M \xrightarrow{t} M'$ iff for all $p \in P$ we have $M(p) < I((p,t))$, $M(p) \geq W((p,t))$, and $M'(p) = M(p) - W((p,t)) + W((t,p))$, thus updating the resulting marking according to the weight function. If $p \in \bullet t$ and $M(p) < W((p,t))$ we say that p *disables* t in M, and if $p \in \circ t$ and $M(p) \geq I((p,t))$, we say that p *inhibits* t in M. In the context of Petri nets, we refer to actions as transitions, and rather than executing actions, we *fire* transitions and call an action sequence for a *firing sequence.*

For a firing sequence $w \in T^*$ where $T = \{t_1, t_2, \dots, t_n\}$, the *Parikh vector* $\wp(w)$ of w is the vector $[\#_{t_1}(w)\ \#_{t_2}(w)\ \dots\ \#_{t_n}(w)]$. The function $\#_t$ is given as $\#_t : T^* \rightarrow \mathbb{N}^0$ and defined recursively as follows (where $a \in T$ and $v \in T^*$):

$$\#_t(\varepsilon) = 0$$
$$\#_t(av) = \begin{cases} 1 + \#_t(v) & \text{if } a = t, \\ \#_t(v) & \text{else} \end{cases}$$

where $\#_t(w)$ is the number of occurrences of t in the firing sequence w.

Example 1 (Graphical Notation). Figure 1a illustrates a Petri net. The net consists of three places drawn as circles and four transitions drawn as squares. The edges are arcs where the arrows corresponds to the regular arcs and the arcs with a circle are inhibitor arcs. Note that by definition there is a weight for every place-transition and transition-place pair in the net. The same applies to inhibitor arcs for all place-transitions pairs. If the weight of a regular arc is 0, then it effectively have no effect on the net and can safely be ignored. The same applies to inhibitor arcs with a weight of infinity. All regular and inhibitor arcs have a weight of 1 unless stated otherwise. The dots inside places denote tokens. For example, the current marking has 2, 0, 1 and 0 tokens for places p_1, p_2, p_3 and p_4, respectively. Fig. 1b illustrates the corresponding LTS of the Petri net in Fig. 1a. The LTS has 8 states drawn as rectangles. Each state is labelled with the marking it represents. We write markings by denoting the place name for each place with at least 1 token. If a place has more than 1 token then it is explicitly stated as a prefix of the place name. For example, the marking $(p_1p_2p_3)$ means that p_1, p_2, and p_3 all have 1 token, and p_4 has 0 tokens. In marking $(2p_1p_3)$,

p_1 has 2 tokens, p_3 has 1 token, and the remaining places have 0 tokens. Edges correspond to the transition relation of the LTS.

2.3 Atomic Propositions for Petri Net CTL

The satisfiability of CTL formulae in a Petri net is interpreted on the LTS generated by the net. We fix the set of atomic propositions AP based on the informal semantics in the MCC Property Language [2], which includes arithmetic expressions and fireability of transitions. Let $N = (P, T, W, I)$ be a Petri net. An atomic proposition $a \in AP$ is defined as:

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

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

where $t \in T$, $c \in \mathbb{N}^0$, $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $p \in P$, and $\oplus \in \{+, -, *\}$. The semantics of φ is defined for a marking M as follows:

$$\begin{aligned} M &\models t && \text{iff } t \in en(M) \\ M &\models e_1 \bowtie e_2 && \text{iff } eval_M(e_1) \bowtie eval_M(e_2) \end{aligned}$$

The semantics of an arithmetic expression in a marking M is given as follows:

$eval_M(c) = c$,
$eval_M(p) = M(p)$,
$eval_M(e_1 \oplus e_2) = eval_M(e_1) \oplus eval_M(e_2)$.

We use $\Phi_{Reach} \subseteq \Phi_{CTL}$ to denote a subset of formulae called *reachability* formulae. Reachability formulae can be on the form $EF\varphi$ or $AG\varphi$, where φ is defined as follows:

$\varphi ::= \ true \mid false \mid a \mid deadlock \mid e_1 \bowtie e_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \varphi_1 \implies \varphi_2 \mid \varphi_1 \iff \varphi_2$

A reachability formula $AG\varphi$ is equivalent to $\neg EF \neg\varphi$. Henceforth, we assume all $AG\varphi$ reachability formulae have been transformed to EF formulae.

For a reachability formula $EF\varphi$ we distinguish between a *cardinality* formula, *fireability* formula, or *deadlock* formula depending on the subexpressions of φ. A cardinality formula excludes the use of t and *deadlock* and compares the number of tokens in places using $e_1 \bowtie e_2$. A fireability formula excludes the use of $e_1 \bowtie e_2$ and *deadlock* and asks if there is a marking where some specific transitions can be fired using t. A deadlock formula is on the form $EF\,deadlock$ and only checks for the existence of deadlocks.

Example 2 (Cardinality Formula). Consider the Petri net in Fig. 1a. We can verify the cardinality formula $(2p_1p_3) \models EF\ p_2 = 2 \wedge p_4 = 1$ by exploring the LTS in Fig. 1b. Doing this, we can conclude that a marking is reachable from the initial marking such that p_2 has 2 tokens and p_4 has 1 token, by for example executing $t_3t_1t_1$, and thus the formula is satisfied.

Example 3. Consider again the Petri net in Figure 1a We can verify the CTL formula $(2p_1p_3) \not\models A(p_1 > 0 \; U \; t_4)$ by exploring the LTS in Fig. 1b. We can fire the sequence t_1t_1 which leaves us in a state where $p_1 > 0$ is not satisfied, but t_4 is not enabled in the resulting marking since t_3 needs to be fired in order to enable t_4. Therefore the formula is not satisfied since there exists a path where it does not hold.

In the remainder of this thesis, we use the following equivalence definition of CTL formulae.

Definition 2. *Let $N = (P, T, W, I)$ be a Petri net, M_0 an initial marking on N, and $\varphi_1, \varphi_2 \in \Phi_{CTL}$ CTL formulae. We write $\varphi_1 \equiv \varphi_2$ iff for all $M \in reach(M_0)$, $M \models \varphi_1$ iff $M \models \varphi_2$.*

2.4 Integer Linear Program

For defining an integer linear program, we first need to define a linear equation. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of variables and $\overline{x}$ a column vector over the variables X such that:

$$\overline{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

A linear equation is given by $\overline{c} \; \overline{x} \bowtie k$, where $\bowtie \in \{=, <, \leq, >, \geq\}$, $k \in \mathbb{Z}$ is an integer, and $\overline{c}$ is a row vector of integers such that:

$$\overline{c} = \begin{bmatrix} c_1 \; c_2 \cdots c_n \end{bmatrix} \quad \text{where } c_i \in \mathbb{Z} \text{ for all } 1 \leq i \leq n.$$

Definition 3 (Integer Linear Program). *An integer linear program $LP = \{\overline{c}_1\overline{x} \bowtie_1 k_1, \overline{c}_2\overline{x} \bowtie_2 k_2, \cdots, \overline{c}_m\overline{x} \bowtie_m k_m\}$ is a set of linear equations. A solution to LP is a mapping $u : X \rightarrow \mathbb{N}^0$ from variables to natural numbers and corresponding column vector $\overline{u}^T = [u(x_1) \; u(x_2) \cdots u(x_n)]$, such that for all $1 \leq i \leq m$ we have $\overline{c}_i\overline{u} \bowtie_i k_i$ is true. We use $\mathcal{E}^X_{lin}$ to denote the set of all linear programs over a set of variables X.*

An integer linear program with a solution is said to be either *feasible* or *infeasible.* For completeness sake, linear programming is also called linear optimisation. It is a technique for optimising some linear objective function while being subject to a number of linear constraints. An example of an objective function is *max* $\overline{vx}$, where $\overline{v}$ is a row vector of reals, including negative reals. The optimisation problem is finding a solution $\overline{c}$ such that $\overline{vc}$ is maximised. We do not need the optimisation aspect for our usage of linear programming, so we omit it and assume the objective function is always *max* $\mathbf{0}\overline{x}$ where $\mathbf{0}$ is a row vector consisting only of zeros, i.e. every solution is an optimal one.

The feasability of linear programs can be decided in polynomial time [8], for example by the use of the simplex algorithm. The feasability of integer linear

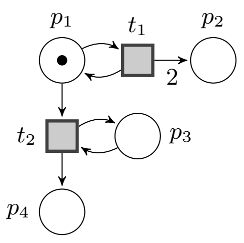


Fig. 2: A Petri net illustrating the state equations technique.

programs, which is the type of program that we use to model our problem, is NP-complete [17,8]. There exist efficient solvers for this type of programs, such as lp_solve [3].

For Petri nets, there is a technique for checking if a marking is unreachable using integer linear programming. This technique is called *state equations*, an algebraic description of how markings change by firing transitions [11].

Let $N = (P, T, W, I)$ be a Petri net, $M_0 \in \mathcal{M}(N)$ an initial marking on N, $M \in \mathcal{M}(N)$ a marking on N, and $X = \{x_t \mid t \in T\}$ a set of variables. From this we construct the following linear program over X:

$$M_0(p) + \sum_{t \in T} (W(t,p) - W(p,t))x_t = M(p) \quad \text{for all } p \in P.$$

It is well-known that if $M \in \mathit{reach}(M_0)$ then there exists a solution to the linear program. If the linear program is infeasible, then we can discern that $M \notin \mathit{reach}(M_0)$. If a solution does exist then it is inconclusive whether or not M is reachable from M_0.

A solution to the state equations is seen as a candidate firing sequence. The natural number assigned to the variable x_t corresponds to the number of times the transition t is fired during the sequence. The reason a solution to the state equation does not decide whether the marking M is reachable is because it does not maintain the Petri net semantics for any intermediary marking between M_0 and M. Transitions that are inhibited or disabled can be fired, and places can have less than zero tokens as long it has the exact number of required tokens after all transitions have been fired. If $w \in T^*$ is a firing sequence such that $M_0 \xrightarrow{w} M$ then $\wp(w)$ is a solution to the linear program, but a solution is not necessarily a possible firing sequence.

Example 4 (State Equations). Consider the Petri net in Figure 2 with initial marking $M_0(p_1) = 1$ and $M_0(p_2) = M_0(p_3) = M_0(p_4) = 0$. We want to check using state equations whether the markings M_1 and M_2 are reachable from M_0 where:

- $M_1(p_1) = 1$, $M_1(p_2) = 5$, $M_1(p_3) = 0$, and $M_1(p_4) = 0$.
- $M_2(p_1) = 0$, $M_2(p_2) = 0$, $M_2(p_3) = 0$, and $M_2(p_4) = 1$.

For M_1 we have the following equations:

$$\begin{aligned}1 + 0x_{t_1} - 1x_{t_2} &= 1\\ 0 + 2x_{t_1} + 0x_{t_2} &= 5\\ 0 + 0x_{t_1} + 0x_{t_2} &= 0\\ 0 + 0x_{t_1} + 1x_{t_2} &= 0\end{aligned}$$

The problem here is the second equation. There does not exist an integer we can assign to x_{t_1} such that the equation is satisfied. We therefore have that $M_1 \notin \mathit{reach}(M_0)$.

For M_1 we have the following equations:

$$\begin{aligned}1 + 0x_{t_1} - 1x_{t_2} &= 0\\ 0 + 2x_{t_1} + 0x_{t_2} &= 0\\ 0 + 0x_{t_1} + 0x_{t_2} &= 0\\ 0 + 0x_{t_1} + 1x_{t_2} &= 1\end{aligned}$$

The assignment $x_{t_1} = 0$ and $x_{t_2} = 1$ is a solution to this integer linear program. However, it is clear from Figure 2 that M_2 is not reachable, since transition t_2 is disabled in M_0. So the set of markings that have a solution using state equations is an over approximation of the actual reachable markings from a given initial marking.

3 Reductions of LTS

State space reduction techniques attempt to reduce the size of the state space at the cost of adding a justified computational overhead. The techniques often have great practical influence, causing model checkers to provide verification answers on model instances where they could not previously provide an answer within reasonable time and resources.

A reduction can be seen as a filter, that filters sets of actions in each state that are required to be executed in order to reach a state satisfying some property. We say that a reduction preserves a set of properties if the properties remain satisfiable in the reduced state space. A reduction is a function from the set of states to the power set of actions, such that for each state the function returns the set of required actions.

Definition 4 (Reduction). *Let $T = (\mathcal{S}, A, \rightarrow)$ be an LTS. A reduction of T is a function $St : \mathcal{S} \rightarrow 2^A$.*

A reduction defines a subset of the transition relation of an LTS, and we annotate the transition relation with a reduction to define the reduced state space.

Definition 5 (Reduced transition relation). *Let $T = (\mathcal{S}, A, \rightarrow)$ be an LTS and St a reduction of T. A reduced transition relation is a relation $\xrightarrow[St]{} \subseteq \rightarrow$ such that $s \xrightarrow[St]{a} s'$ iff $s \xrightarrow{a} s'$ and $a \in St(s)$.*

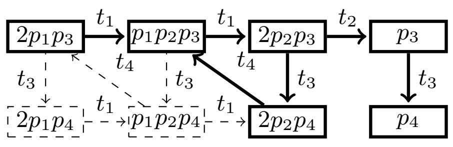


Fig. 3: Example reduced state space for the Petri net in Figure 2.

Let $T = (\mathcal{S}, A, \rightarrow)$ be an LTS, $a \in \mathcal{S}$ a state, and St a reduction of T. The set $\overline{St(s)} = A \setminus St(s)$, is the set of all actions not in $St(s)$.

For a sequences of actions, the following condition identifies required actions, that allow us to permute the sequence, such that the permuted sequence begins with the required action.

W For all $s \in \mathcal{S}$, all $a \in St(s)$, and all $w \in \overline{St(s)}^*$, if $s \xrightarrow{wa} s'$ then $s \xrightarrow{aw} s'$.

Reductions that satisfy **W** are called *(weak)semistubborn* reductions [24], and for all $s \in \mathcal{S}$, we say that $St(s)$ is the *stubborn set* of s, and that an action $a \in St(s)$ is a *stubborn action.*

W implies that if a is enabled after a sequence of non-stubborn actions leading from s to s', then the stubborn action can be moved to the front of the sequence, without disabling the non-stubborn transitions. If a is disabled in s, then it remains disabled after the execution of any sequence of non-stubborn actions.

Lemma 1. *Let $T = (\mathcal{S}, A, \rightarrow)$ be an LTS and St be a reduction on T satisfying **W**. For all $s \in \mathcal{S}$, all $a \in St(s)$, and all $w \in \overline{St(s)}^*$, if $a \notin en(s)$ and $s \xrightarrow{w} s'$ then $a \notin en(s')$.*

Proof. The proof proceeds by contradiction.

Let $s, s' \in \mathcal{S}$, $a \in St(s)$, $a \notin en(s)$, and $w \in \overline{St(s)}^*$ s.t. $s \xrightarrow{w} s'$. Assume that $a \in en(s')$. If $a \in en(s')$ then we must have that $a \in en(s)$ by **W**, which contradicts that $a \notin en(s)$. Therefore, $a \notin en(s')$. □

Example 5 (Stubborn Reduction). Consider the Petri net in Figure 1a. We want to create a stubborn reduction such that its corresponding state space in Figure 1b is reduced. Consider the set $\{t_1\}$ in the initial state $(2p_1p_3)$. There is only one non-stubborn transition sequence from $(2p_1p_3)$, namely t_3, where t_1 is enabled in the resulting marking, so the firing sequence t_3t_1 is possible. Therefore $\{t_1\}$ is a stubborn set for the initial state. If we instead consider the set $\{t_3\}$, it is not a stubborn set for the initial state. This is because there exist the non-stubborn firing sequence $t_1t_1t_2$ followed by t_3, but the firing sequence $t_3t_1t_1t_2$ is not possible. A possible reduced state space satisfying **W** for Figure 1a can be seen in Figure 3. In fact, the state space consisting only of the initial state $(2p_1p_3)$ is a valid reduction, since the empty set is always a stubborn set satisfying **W**.

3.1 Reachability Preserving Stubborn Reduction

When performing reachability analysis, we are searching for states that satisfy a given property. In the context of stubborn reduction, we refer to these states as goal states. The task of investigating whether a goal state is reachable from an initial state is referred to as a *reachability problem.*

Let $T = (\mathcal{S}, A, \rightarrow)$ be an LTS, $s_0 \in \mathcal{S}$ an initial state, and $G \subseteq \mathcal{S}$ a set of goal states. For a reduction St to preserve paths to a goal state, the following condition needs to be satisfied:

R For all $s \in \mathcal{S}$ if $s \notin G$ and $s \xrightarrow{w} s'$ where $w \in \overline{St(s)}^*$ then $s' \notin G$.

Rule **R** states that, when starting in a non-goal state, the execution of non-stubborn transitions cannot reach any goal state in G. It also ensures that at least one stubborn action has to be executed in order to reach a goal state.

The following theorem ensures that for any path from an initial state to a goal state, there exists a path in the reduced transition relation leading to the same goal state in the same number of steps, or a different goal state in at most the same number of steps.

Theorem 1 (Reachability preservation). *Let $(\mathcal{S}, A, \rightarrow)$ be an LTS, $G \subseteq S$ a set of goal states, and $s_0 \in \mathcal{S}$. Let St be a reduction satisfying **W** and **R**. If $s_0 \rightarrow^n s$ where $s \in G$ then $s_0 \underset{St}{\longrightarrow}^m s'$ where $s' \in G$ and $m \leq n$. If $s_0 \underset{St}{\longrightarrow}^m s$ where $s \in G$ then $s_0 \rightarrow^m s$.*

Proof. The second part of the theorem is trivial since $\underset{St}{\longrightarrow} \subseteq \rightarrow$.

The proof for the first part of the theorem proceeds by induction on n, for the induction hypothesis if $s_0 \rightarrow^n s$ where $s \in G$ then $s_0 \underset{St}{\longrightarrow}^m s'$ where $s' \in G$ and $m \leq n$. Let $w \in A^*$ be a transition sequence of length n, such that $s_0 \xrightarrow{w} s$ for $s \in G$.

Base step: If $n = 0$, then $s_0 = s$ and the induction hypothesis holds.

Induction step: If $n > 0$ and $s_0 \notin G$, then by **R** if none of the actions in w is in $St(s_0)$, then **R** implies that $s \notin G$, contradicting our assumption that s is a goal state. We must therefore have that at least one transition in w is stubborn, and we can divide w into vau, where $v \in \overline{St(s_0)}^*$ and $a \in St(s_0)$. Rule **W** implies the existence of s'' such that $s_0 \xrightarrow{a} s'' \xrightarrow{vu} s$. If $s'' \in G$, the length of the path from s_0 to s'' is less than n and we are done. Otherwise, there exists a path $s'' \xrightarrow{vu} s$ of length $n-1$ from s'' to s. By the induction hypothesis, $s'' \underset{St}{\longrightarrow}^{m'} s'$ where $m' \leq n-1$ which together with $s_0 \underset{St}{\xrightarrow{a}} s''$ gives $s_0 \underset{St}{\longrightarrow}^{m'+1} s'$ where $m' + 1 \leq n$. □

4 Reductions of Petri Net

We now show how to do reachability preserving stubborn reduction for Petri net with inhibitor arcs. Instead of states and actions of LTS, we now refer to markings and transitions of Petri nets. It is inefficient to generate stubborn sets for markings using **W** and **R** directly, as the properties must hold for even infinite transition sequences, and ensuring these properties is no different from performing exhaustive state space search. So rather than examining the entire state space, we now analyse the structure of the Petri net and present algorithms for iteratively constructing the stubborn sets, based on techniques in [15].

We define goal states as goal markings that satisfy a given reachability property. Let $EF\ \varphi \in \Phi_{Reach}$ be a reachability formula and $G_\varphi = \{M \in \mathcal{M}(N) \mid M \models \varphi\}$ be the goal markings for φ, where N is a Petri net. The reduction procedure must identify transitions that are required to fire in order to reach the goal markings. To identify these transitions, we update the notion of attractor sets [20] to our reachability logic. We call this set the *interesting transitions* of a marking M and formula φ, denoted $A_M(\varphi)$. All transitions that can alter the truth value of φ from *false* to *true* are interesting transitions.

Assume $M \not\models \varphi$ and $t \in T$. Let $A_M(\varphi) \subseteq T$ such that if $M \xrightarrow{t} M'$ and $M' \models \varphi$ then $t \in A_M(\varphi)$. We define $A_M(\varphi)$ recursively on the syntactic category for reachability formulae. The interesting transitions for all Boolean formulae are shown in Table 1. The interesting transitions of a negation depend on what follows syntactically from the negation, and thus we describe this in a separate column. Table 1 does not describe $A_M(\neg\neg\varphi)$ because its set of interesting transitions is equivalent to that of $A_M(\varphi)$. We introduce the notation $\overline{\bowtie}$ that refers to the complement of of a comparison operator $\bowtie$. The complement operators are shown in Table 2.

The interesting transitions of a formula $e_1 \bowtie e_2$ depend on $\bowtie$ and the marking of places found in the formula.

Example 6 (Interesting Transitions for Comparisons). Consider $\varphi = p > 5$ where $p \in P$ and $M(p) = 4$. As there are currently not enough tokens in p, we must fire transitions that produce tokens into p, i.e. $\bullet p \subseteq A_M(\varphi)$. Now consider $\varphi = (8-p) > 5$. Reducing the token count of p will increase the evaluation of $(8-p)$, and thus we need the transitions that consume tokens from p, i.e. $p\bullet \subseteq A_M(\varphi)$.

We define the set of expressions that can be constructed with N as E_N, and two functions $incr_M : E_N \to 2^T$ and $decr_M : E_N \to 2^T$. These functions receive an expression e and return the set of transitions that, when fired, increase and decrease the evaluation of e, respectively. We present the interesting transitions for formulae of the form $e_1 \bowtie e_2$ in Table 3. We recursively define $incr_M$ and $decr_M$ on the syntax of expressions in Table 4.

Example 7 (Interesting Set of Transitions). Consider the Petri net in Figure 1a with initial marking $M_0 = (2p_1p_3)$. We want to create an interesting set of transitions for the reachability formula $EFp_2 = 2\ \wedge\ p_4 = 1$. Both of the places p_2 and p_4 have less than the required tokens, so the interesting set of trantitions are the transitions that increase the evaluation of both expressions. From this we have that $A_{M_0}(p_2 = 2) = \{t_1\}$ and $A_{M_0}(p_4 = 1) = \{t_3\}$. In Table 1 we can see that the interesting set of transitions for a conjunction is the interesting

Formula φ	$A_M(\varphi)$	$A_M(\neg\varphi)$
true	$\emptyset$	$\emptyset$
false	$\emptyset$	$\emptyset$
t	$\bullet p$ for some $p \in \bullet t$ where $M(p) < W(p,t)$ or $p\bullet$ for some $p \in \circ t$ where $M(p) \geq I(p,t)$	$(\bullet t)\bullet \cup \bullet(\circ t)$
deadlock	$(\bullet t)\bullet \cup \bullet(\circ t)$ for some $t \in en(M)$	$\emptyset$
$e_1 \bowtie e_2$	See Table 3	$A_M(e_1 \,\overline{\bowtie}\, e_2)$
$\varphi_1 \wedge \varphi_2$	$A_M(\varphi_i)$ for some $i \in \{1,2\}$ where $M \not\models \varphi_i$	$A_M(\neg\varphi_1 \vee \neg\varphi_2)$
$\varphi_1 \vee \varphi_2$	$A_M(\varphi_1) \cup A_M(\varphi_2)$	$A_M(\neg\varphi_1 \wedge \neg\varphi_2)$
$\varphi_1 \implies \varphi_2$	$A_M(\neg\varphi_1 \vee \varphi_2)$	$A_M(\varphi_1 \wedge \neg\varphi_2)$
$\varphi_1 \iff \varphi_2$	$A_M(\varphi_1 \implies \varphi_2 \wedge \varphi_2 \implies \varphi_1)$	$A_M(\varphi_1 \iff \neg\varphi_2)$

Table 1: Interesting transitions of φ.

Operator $\bowtie$	$\overline{\bowtie}$
$<$	$\geq$
$\leq$	$>$
$=$	$\neq$
$\neq$	$=$
$>$	$\leq$
$\geq$	$<$

Table 2: Complement of comparison operator $\bowtie$.

Formula $e_1 \bowtie e_2$	$A_M(e_1 \bowtie e_2)$
$e_1 < e_2$	$decr_M(e_1) \cup incr_M(e_2)$
$e_1 \leq e_2$	$decr_M(e_1) \cup incr_M(e_2)$
$e_1 > e_2$	$incr_M(e_1) \cup decr_M(e_2)$
$e_1 \geq e_2$	$incr_M(e_1) \cup decr_M(e_2)$
$e_1 = e_2$	if $eval_M(e_1) > eval_M(e_2)$ then $decr_M(e_1) \cup incr_M(e_2)$ else if $eval_M(e_1) < eval_M(e_2)$ then $incr_M(e_1) \cup decr_M(e_2)$
$e_1 \neq e_2$	$incr_M(e_1) \cup decr_M(e_1) \cup incr_M(e_2) \cup decr_M(e_2)$

Table 3: Interesting transitions of $e_1 \bowtie e_2$.

Expression e	$incr_M(e)$	$decr_M(e)$
c	$\emptyset$	$\emptyset$
p	$\bullet p$	$p\bullet$
$e_1 + e_2$	$incr_M(e_1) \cup incr_M(e_2)$	$decr_M(e_1) \cup decr_M(e_2)$
$e_1 - e_2$	$incr_M(e_1) \cup decr_M(e_2)$	$decr_M(e_1) \cup incr_M(e_2)$
$e_1 * e_2$	$incr_M(e_1) \cup decr_M(e_1) \cup incr_M(e_2) \cup decr_M(e_2)$	$incr_M(e_1) \cup decr_M(e_1) \cup incr_M(e_2) \cup decr_M(e_2)$

Table 4: Increasing and decreasing transitions of e.

set of transitions for one of its children that is not satisfied in M_0. So e.g. $A_{M_0}(p_2 = 2 \wedge p_4 = 1) = \{t_1\}$ is a valid set of interesting transitions. The other option is $A_{M_0}(p_2 = 2 \wedge p_4 = 1) = \{t_3\}$.

Given a marking and an unsatisfied reachability formula, Lemma 2 ensures that if a non-interesting transition is fired, we will not reach a marking that satisfies the given formula. Consequently, if we reach a marking where the formula is satisfied, the fired transition is necessarily an interesting transition.

Lemma 2. *Let $N = (P, T, W, I)$ be a Petri net, $M \in \mathcal{M}(N)$ a marking, and $EF\varphi \in \Phi_{Reach}$ a reachability formula. If $M \not\models \varphi$ and $M \xrightarrow{t'} M'$ where $t' \notin A_M(\varphi)$ then $M' \not\models \varphi$.*

Proof. The proof proceeds by structural induction of φ.
Base step:

- $\varphi = true$: The assumption does not hold, so there is nothing to show.
- $\varphi = false$: Clearly $M' \not\models \varphi$ holds.
- $\varphi = t$: Since $M \not\models \varphi$ we have t is either disabled, inhibited, or both in M. If t is disabled, then $\bullet p \subseteq A_M(\varphi)$ for some $p \in \bullet t$ where $M(p) < W(p,t)$. We know $t' \notin A_M(\varphi)$, so t' cannot be any transition that puts tokens into p. Because of this, we must have that $M(p) \geq M'(p)$ and t is disabled in M'. If t is inhibited, then $p\bullet \subseteq A_M(\varphi)$ for some $p \in \circ t$ where $M(p) \geq I(p,t)$. We know $t' \notin A_M(\varphi)$, so t' cannot be any transition that subtracts tokens from p. Because of this, we must have that $M(p) \leq M'(p)$ and t is inhibited in M'. Therefore, after firing t' we have that t is still not enabled, so $M' \not\models \varphi$ holds.
- $\varphi = deadlock$: Since $M \not\models \varphi$ we know M is not a deadlock and there exists $t \in en(M)$. We know that $(\bullet t)\bullet \subseteq A_M(\varphi)$, so t' cannot remove tokens from the pre-set of t, i.e. for all $p \in \bullet t$ we have $M(p) \leq M'(p)$. We also know that $\bullet(\circ t) \subseteq A_M$, so t' cannot put tokens in the inhibitor pre-set of t, i.e. for all $p \in \circ t$ we have $M(p) \geq M'(p)$. Therefore, after firing t' we have that t is still enabled, so $M' \not\models \varphi$ holds.
- $\varphi = e_1 < e_2$: Since $M \not\models \varphi$ we know that $decr_M(e_1) \cup incr_M(e_2) \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' cannot decrease the evaluation of e_1 or increase the evaluation of e_2, i.e. $eval_M(e_1) \leq eval_{M'}(e_1)$ and $eval_M(e_2) \geq eval_{M'}(e_2)$. So $M' \not\models \varphi$ holds.
- $\varphi = e_1 \leq e_2$: Similar to the proof for $e_1 < e_2$.
- $\varphi = e_1 \neq e_2$: Since $M \not\models \varphi$ we know that $incr_M(e_1) \cup decr_M(e_1) \cup incr_M(e_2) \cup decr_M(e_2) \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' cannot change the evalution of e_1 or e_2 at all, i.e. $eval_M(e_1) = eval_{M'}(e_1)$ and $eval_M(e_2) = eval_{M'}(e_2)$. So $M' \not\models \varphi$ holds.
- $\varphi = e_1 = e_2$: Since $M \not\models \varphi$ we know either $M \models e_1 > e_2$ or $M \models e_1 < e_2$ are true. In the first case, we have that $decr_M(e_1) \cup incr_M(e_2) \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' cannot decrease the evaluation of e_1 or increase the evaluation of e_2, i.e. $eval_M(e_1) \leq eval_{M'}(e_1)$ and $eval_M(e_2) \geq eval_{M'}(e_2)$. In the second case, we have that $incr_M(e_1) \cup decr_M(e_2) \subseteq$

$A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' cannot increase the evaluation of e_1 or decrease the evaluation of e_2, i.e. $eval_M(e_1) \geq eval_{M'}(e_1)$ and $eval_M(e_2) \leq eval_{M'}(e_2)$. So $M' \not\models \varphi$ holds.

$\varphi = e_1 > e_2$: Similar to the proof for $e_1 < e_2$.

$\varphi = e_1 \geq e_2$: Similar to the proof for $e_1 < e_2$.

We now proceed with the inductive cases, and prove them using structural induction.

Inductive step:

$\varphi = \varphi_1 \wedge \varphi_2$: Since $M \not\models \varphi$ we know one of $M \not\models \varphi_1$ or $M \not\models \varphi_2$ are true. Let $i \in \{1, 2\}$ such that $M \not\models \varphi_i$, then we know $A_M(\varphi_i) \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' by the induction hypothesis cannot make φ_i true. So $M' \not\models \varphi$ holds.

$\varphi = \varphi_1 \vee \varphi_2$: Since $M \not\models \varphi$ we know $M \not\models \varphi_1$ and $M \not\models \varphi_2$ are true and $A_M(\varphi_1) \cup A_M(\varphi_2) \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so firing t' by the induction hypothesis cannot make φ_1 or φ_2 true. So $M' \not\models \varphi$ holds.

$\varphi = \neg true$: Trivial.

$\varphi = \neg false$: Trivial.

$\varphi = \neg t$: Since $M \not\models \varphi$ we know that $t \in en(M)$ and $(\bullet t)\bullet \subseteq A_M(\varphi)$. We know $t' \notin A_M(\varphi)$, so t' cannot be any transition that removes tokens from any place in the pre-set of t. Therefore, after firing t' we have that t is still enabled, so $M' \not\models \varphi$ holds.

$\varphi = \neg deadlock$: Since $M \not\models \varphi$ we know that M is a deadlock. There can therefore not exist a t' such that $M \xrightarrow{t'} M'$, and there is nothing to show.

For formulae on the form $\neg(e_1 \bowtie e_2)$ the proof is trivial, since the negation changes the comparison operator into its dual comparison operator, as seen in Table 2, which we have already proved previously.

$\varphi = \neg(\varphi_1 \wedge \varphi_2)$: We know $\neg(\varphi_1 \wedge \varphi_2)$ is equaivalent to $\neg\varphi_1 \vee \neg\varphi_2$, which we have already proved.

$\varphi = \neg(\varphi_1 \vee \varphi_2)$: We know $\neg(\varphi_1 \vee \varphi_2)$ is equaivalent to $\neg\varphi_1 \wedge \neg\varphi_2$, which we have already proved.

$\varphi = \varphi_1 \implies \varphi_2$: We know $\varphi_1 \implies \varphi_2$ is equivalent to $\neg\varphi_1 \vee \varphi_2$, which we have already proved previously.

$\varphi = \neg(\varphi_1 \implies \varphi_2)$: We know $\neg(\varphi_1 \implies \varphi_2)$ is equivalent to $\varphi_1 \wedge \neg\varphi_2$, which we have already proved previously.

$\varphi = \varphi_1 \iff \varphi_2$: We know $\varphi_1 \iff \varphi_2$ is equivalent to $(\varphi_1 \implies \varphi_2) \wedge (\varphi_2 \implies \varphi_1)$, which we have already proved previously.

$\varphi = \neg(\varphi_1 \iff \varphi_2)$: We know $\neg(\varphi_1 \iff \varphi_2)$ is equivalent to $\varphi_1 \iff \neg\varphi_2$, which we have already proved previously. □

With Lemma 2 we can now prove Lemma 3, which guarantees that by firing a sequence of non-interesting transitions, we cannot reach a goal marking.

Lemma 3. *Let $N = (P, T, W, I)$ be a Petri net, $M \in \mathcal{M}(N)$ a marking, φ a formula, and $w \in \overline{A_M(\varphi)}^*$ a sequence of non-interesting transitions. If $M \notin G_\varphi$ and $M \xrightarrow{w} M'$ then $M' \notin G_\varphi$.*

Proof. The proof proceeds by induction on the length n of w, for the induction hypothesis if $M \notin G_\varphi$ and $M \xrightarrow{w} M'$ then we have $M' \notin G_\varphi$.

Base step: Let $n = 0$, then $M = M'$ and we are done.

Induction step: Let $n > 0$. We can divide w into tw' such that $M \xrightarrow{t} M'' \xrightarrow{w'} M'$ and the length of w' is $n - 1$. By Lemma 2 we know that firing t cannot make φ true, so we must have that $M'' \notin G_\varphi$. By the induction hypothesis, we must have that $M' \notin G_\varphi$. $\square$

Lemma 3 corresponds to **R**. We can easily verify the **R** property by including all interesting transitions in the stubborn set. Ensuring the **W** property is done by examining the structure of the Petri net and the marking in question.

Proposition 1 (Reachability preserving closure for Petri nets). *Let $N = (P, T, W, I)$ be a Petri net with inhibitor arcs, $EF\varphi \in \Phi_{Reach}$ a reachability formula, and St a reduction such that for all $M \in \mathcal{M}(N)$:*

1 $A_M(\varphi) \subseteq St(M)$.
2 For all $t \in St(M)$, if $t \notin en(M)$ then
  - *exists p that disables t in M and $\bullet p \subseteq St(M)$, or*
  - *exists p that inhibits t in M and $p\bullet \subseteq St(M)$.*

3 For all $t \in St(M)$, if $t \in en(M)$ then
  - *$(\bullet t)\bullet \subseteq St(M)$, and*
  - *$(t\bullet)\circ \subseteq St(M)$.*

*then St satisfies **W** and **R**.*

Proof.
(**R**): Follows from Lemma 3 by Condition 1.
(**W**): Let $M \in \mathcal{M}(N)$ be a marking, $t \in T$ a transition such that $t \in St(M)$, and $w \in \overline{St(M)}^*$ a transition sequence. We show that if $M \xrightarrow{wt} M'$ then $M \xrightarrow{tw} M'$.
If $t \notin en(M)$, then there exists p that disables or inhibits t in M. Due to condition 2, if p disables t in M then all $t' \in \bullet p$ are also in $St(M)$, thus t' cannot be in w. Because of this we cannot add any tokens to p by firing w and p still disables t after firing w. If p inhibits t in M then all $t' \in p\bullet$ are also in $St(M)$, thus t' cannot be in w. Because of this we cannot remove any tokens from p by firing w and p still inhibits t after firing w. This contradicts our assumption that t is enabled after firing w, hence we can infer that $t \in en(M)$. This completes the first part of **W** where we swap t to the beginning of the transition sequence leading to M', and implies the existence of M'' such that $M \xrightarrow{t} M''$.
The only thing left to show is that we can fire w in M'' and complete the path to M', i.e. $M'' \xrightarrow{w} M'$. If we cannot fire w in M'', then there are two possible cases: There exists $t' \in (\bullet t)\bullet$ that occurs in w and becomes disabled by firing t, or exists $t' \in (t\bullet)\circ$ that occurs in w and becomes inhibited by firing t. In the first case, due to condition 3, we have that $(\bullet t)\bullet \subseteq St(M)$, and t' cannot be in w since it is in the stubborn set. Therefore, w is enabled

Algorithm 1: Construction of a reachability preserving stubborn set

```
input  : $N = (P, T, W, I)$, $M \in \mathcal{M}(N)$, $\varphi$
output : $St(M)$ where $St$ satisfies $\mathbf{W}$ and $\mathbf{R}$
1  $X := \emptyset$; $unprocessed := A_M(\varphi)$;
2  while $unprocessed \neq \emptyset$ do
3      pick any $t \in unprocessed$;
4      if $t \notin en(M)$ then
5          if $Exists\ p \in \bullet t\ s.t.\ M(p) < W(p, t)$ then
6              pick any $p \in \bullet t$ s.t. $M(p) < W(p, t)$;
7              $unprocessed := unprocessed \cup (\bullet p \setminus X)$;
8          else
9              pick any $p \in \circ t$ s.t. $M(p) \geq I(p, t)$;
10             $unprocessed := unprocessed \cup (p \bullet \setminus X)$;
11     else
12         $unprocessed := unprocessed \cup ((\bullet t) \bullet \setminus X) \cup ((t \bullet) \circ \setminus X)$;
13     $unprocessed := unprocessed \setminus \{t\}$;
14     $X := X \cup \{t\}$;
15 return $X$;
```

in M'' and $M'' \xrightarrow{w} M'$. In the second case we have that $(t\bullet)\circ \subseteq St(M)$, and t' cannot be in w since it is in the stubborn set. Therefore, w is not inhibited in M'' and $M'' \xrightarrow{w} M'$.

□

In Algorithm 1 we illustrate pseudocode on how to construct a reachability preserving stubborn set that satisfies **W** and **R** for a given marking M and reachability formula $EF\varphi \in \Phi_{Reach}$.

Lemma 4. *Algorithm 1 terminates.*

Proof. If $A_M(\varphi) = \emptyset$ then we never enter the while-loop and terminate immediately. If $A_M(\varphi) \neq \emptyset$ the loop condition holds and we enter the while-loop. Notice that $X \cap unprocessed = \emptyset$ is always the case in the execution of Algorithm 1. Clearly, we never remove transitions from X after they have been added. Therefore, since in line 14 a new transition is added to X at the end of each loop iteration, the while-loop can iterate at most once for each transition. Since T is finite by the Petri net definition, the while-loop iterates a finite number of times, and Algorithm 1 terminates. □

Lemma 5. *When Algorithm 1 terminates, the reduction St computed by the algorithm satisfies **W** and **R**.*

Proof. The proof proceeds by showing for any marking M that X fulfils every condition from Proposition 1.

Condition 1 clearly holds since the set of interesting transitions are added to the pre-processing set in line 1. Every element in the pre-processing set are eventuualy added to X, so condition 1 holds.

To show condition 2 and condition 3 we need to prove the following loop invariant: For all $t \in T$:

- If $t \notin en(M)$ and $t \in X$ then we have $\bullet p \subseteq \mathit{unprocessed} \cup X$ for some p that disables t in M or $p\bullet \subseteq \mathit{unprocessed} \cup X$ for some p that inhibits t in M.
- If $t \in en(M) \cap X$ then $(\bullet t)\bullet \subseteq \mathit{unprocessed} \cup X$ and $(t\bullet)\circ \subseteq \mathit{unprocessed} \cup X$.

Initialisation: Prior to the while-loop $X = \emptyset$ so the loop invariant holds.

Maintenance: Assume the loop invariant holds after the nth iteration of the while-loop. Let $t \in \mathit{unprocessed}$ be the selected transition for iteration $n + 1$ in line 3. If $t \notin en(M)$ there are two cases:

- If p disables t in M then for all $t' \in \bullet p$ either $t' \in \mathit{unprocessed}$ or $t' \in X$ due to line 6 and line 7.
- If p inhibits t in M then for all $t' \in p\bullet$ either $t' \in \mathit{unprocessed}$ or $t' \in X$ due to line 9 and line 10.

If $t \in en(M)$ then for all $t' \in (\bullet t) \bullet \cup (t\bullet)\circ$ either $t' \in \mathit{unprocessed}$ or $t' \in X$ due to line 12. So the loop invariant holds for the next iteration of the while-loop.

Termination: When $\mathit{unprocessed} = \emptyset$, for all $t \in X$ we have t either fulfils the first or second part of the loop invariant, and it clearly holds. Since $\mathit{unprocessed} = \emptyset$, we must have that all the transitions that ensure the loop invariant is fulfilled must be in X. Because of this, X fulfills condition 2 and condition 3 of Proposition 1, and X is a stubborn set for M satisfying $\mathbf{W}$ and $\mathbf{R}$. □

Example 8 (Stubborn Set Generation). Consider the Petri net in Figure 1a with initial marking $M_0 = (2p_1p_3)$. We want to generate a stubborn set for M_0 and the reachability formula $\mathit{EF}\ p_2 = 2 \wedge p_4 = 1$ satisfying $\mathbf{W}$ and $\mathbf{R}$. From Example 7 we know that both $\{t_1\}$ and $\{t_3\}$ are valid interesting sets of transitions for $\mathit{EF}\ p_2 = 2 \wedge p_4 = 1$ in M_0. Let $\{t_1\}$ be the set of transitions added to *unprocessed* in line 1 of Algorithm 1. When the while-loop is entered, we select t_1 in line 3. The transition is enabled so the sets $(\bullet t_1)\bullet$ and $(t\bullet)\circ$ are added to X. It is clear that $(\bullet t_1)\bullet = \{t_1\}$ and $(t_1\bullet)\circ = \emptyset$, so nothing is added to *unprocessed* at all, and the algorithms terminates when t_1 is added to X. So $St(M_0) = \{t_1\}$ is a valid stubborn set for M_0. If we instead choose $\{t_3\}$ as the interesting set of transitions, we have that $(\bullet t_3)\bullet = \{t_3\}$ and $(t_3\bullet)\circ = \{t_2\}$, and t_2 is added to *unprocessed*. In the next iteration we choose t_2 in line 3 and add t_1 to *unprocessed* in line 6 since t_2 is inhibited in M_0 by the place p_2, and $\bullet p_2 = \{t_1\}$. In the iteration for t_1 we do not add anything to *unprocessed* and the algorithms terminates. So $St(M_0) = \{t_1, t_2, t_3\}$ is also a valid stubborn set for M_0, and mirrors the case in Example 5 when we initially choose transition t_3.

As demonstrated in example 8, there exists nondeterminism in both generating the interesting set of transitions and applying the stubborn set closure. For the interesting set of transitions, this is whenever we have conjunction and deadlock formulae. For the stubborn set closure, it is whenever we have to choose either a disabling or inhibiting place for disabled transitions. We often prefer a smaller stubborn set, but this is not guaranteed to be more efficient in terms of state space reduction, as demonstrated in [26]. If the choice is between two stubborn sets, neither of which is a subset of the other, then it cannot be discerned without exploring the state space which one is the best choice, even if one of them has fewer elements.

5 The Siphon-Trap Property

Stubborn reduction is used while performing explicit state space analysis, and is therefore subject to the the state space explosion. Because of this, techniques that analyze the structural properties of Petri net are promising, as the number of places and transitions is finite and substantially smaller than the size of the state space

It is possible to check for deadlock freedom in Petri nets by examining structural entities within a Petri net called *siphons* and *traps*. For this we only consider 1-weighted Petri nets without inhibitor arcs. A Petri net $N = (P, T, W, I)$ is 1-weighted if $W : (P \times T) \cup (T \times P) \to \{0, 1\}$, i.e. every regular arc have a weight of 0 or 1. N have no inhibitor arcs if for all $p \in P$ and $t \in T$ we have $I(p, t) = \infty$, i.e. the inhibitor arcs have no effect on the enabledness of the transitions.

Definition 6 (Siphon). *Let $N = (P, T, W, I)$ be a 1-weighted Petri net with no inhibitor arcs and M_0 an initial marking on N. A siphon D of N, is a non-empty set of places $D \subseteq P$, where $\bullet D \subseteq D\bullet$. We say that D is marked if there exists a place $p \in D$ with $M_0(p) > 0$.*

Definition 7 (Trap). *Let $N = (P, T, W, I)$ be a 1-weighted Petri net with no inhibitor arcs and M_0 an initial marking on N. A trap Q of N, is a non-empty set of places $Q \subseteq P$, where $Q\bullet \subseteq \bullet Q$. We say that Q is marked if there exists a place $p \in Q$ with $M_0(p) > 0$.*

Intuitively, a siphon cannot go from being unmarked to marked, and a trap cannot go from being marked to unmarked.

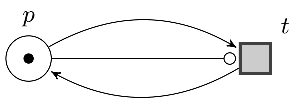
The property in question is called the *siphon-trap property* and forms a relation between the siphons and traps in a Petri net, and ensures deadlock freedom if it is satisfied. The siphon-trap property for a Petri net requires that every siphon contains a marked trap and is defined as follows:

Definition 8 (Siphon-Trap Property). *Let $N = (P, T, W, I)$ be a 1-weighted Petri net with no inhibitor arcs and M_0 an initial marking on N. We say that N has the siphon-trap property if for every siphon $D \subseteq P$ there exists a trap $Q \subseteq D$ s.t. Q is marked.*

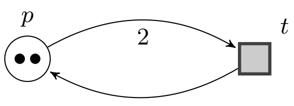
Now we present the Commoner-Hack property that correlates the siphon-trap property with deadlock freedom [9].

Proposition 2 (Commoner-Hack). *Let N be a 1-weighted Petri net with no inhibitor arcs and M_0 an initial marking on N. If N has the siphon-trap property then no deadlock is reachable from M_0.*

Example 9 (Siphon Trap Counterexamples). Consider the Petri net in Figure 4a. The set $\{p\}$ is the only siphon in the Petri net and it is also its corresponding



(a) A Petri net illustrating why inhibitor arcs does not work in the siphon-trap property.



(b) A Petri net illustrating why arc weights does not work in the siphon-trap property.

marked trap, so by Commoner-Hack the Petri net does not have a deadlock. However, clearly the transition t is not enabled since it is being inhibited by p, and therefore the net is not deadlock free. This contradicts the Commoner-Hack property. Now consider the Petri net in Figure 4b. It is a not a 1-weighted Petri net since the weight of the arc from p to t have a weight of 2. The set $\{p\}$ is the only siphon in the Petri net and it is also its corresponding marked trap, so by Commoner-Hack the Petri net does not have a deadlock. However, after firing t we are left with one token in p and results in p disabling t, creating a deadlock and contradicting the Commoner-Hack property.

5.1 Siphon-Trap Property Using Integer Linear Programming

Let $N = (P, T, W, I)$ be a 1-weighted Petri net with no inhibitor arcs and M_0 an initial marking on N. We know that N has the siphon-trap property if for every siphon $D \subseteq P$ there exists a trap $Q \subseteq D$ s.t. Q is marked. Let D be a siphon of N. The unique maximal trap of D is the union of all traps within D, written Q_{max} where traps are closed under union. Clearly, there exists a marked trap within D if and only if the maximal trap Q_{max} of D is marked. There can potentially be exponentially many traps within a siphon, so for proving the siphon-trap property we want to focus on finding a specific representative trap instead, which is the maximal trap. From this we can convert the problem into an appropriate form:

$$\begin{aligned} \text{for all siphons } D \subseteq P \ \text{ exists a trap } Q \subseteq D \text{ s.t. } Q \text{ is marked} &\iff \\ \neg(\text{exists a siphon } D \subseteq P \ \text{ s.t. for all traps } Q \subseteq D \ \textit{s.t.}\ Q \text{ is not marked}) &\iff \\ \neg(\text{exists a siphon } D \text{ s.t. the maximal trap } Q_{max} \text{ of } D \text{ is not marked}) & \end{aligned}$$

We want to prove that there exists a siphon whose maximal trap is not marked in order to disprove the siphon-trap property. If we cannot prove this, then the Petri net must have the siphon-trap property. Otherwise, if it is not possible then it cannot be decided whether or not the Petri net has the siphon-trap property or not. In [18] the authors characterise this as a Boolean satisfiability problem (SAT). The procedure starts by selecting a siphon and then iteratively removing places from the siphon until we have either the unique maximal trap within the siphon or the empty set. This iteration is bounded since the number of places in a Petri net is finite.

We characterise this problem as an integer linear program, such that the generated program has a solution if and only if there exists a siphon which does not contain a marked trap. Thus, we search for a solution that disproofs the siphon-trap property. We parameterise the procedure with a depth, for how many iterations of place removal, before we give up on finding the maximal trap. This puts bound both the number of decision variables and constraints generated in the procedure. The reason for this bound is to optimise resources used for deciding the siphon-trap property, as the time needed to generate the integer linear program and solving it is exponential.

Let $N = (P, T, W, I)$ be a 1-weighted Petri net with no inhibitor arcs, M_0 an initial marking on N, and $d \in \mathbb{N}^0$ a natural number indicating the depth of the procedure. We have a sequence of sets $X_0, X_1, \ldots, X_d$ such that:

$$P \supseteq X_0 \supseteq X_1 \supseteq \ldots \supseteq X_d$$

The set X_0 represents the initially selected siphon and each subsequent set represents a candidate maximal trap for the siphon, moving towards either the maximal trap or the empty set. For each place p we have $d+1$ decision variables such that for all $0 \leq i \leq d$ we have $p^i \in \{0,1\}$, and $p^i = 1$ if and only if $p \in X_i$.

Additionally, we introduce $d+1$ decision variables for each transition t, written as $post_t^d$, such that for all $0 \leq i \leq d$ we have $post_t^i \in \{0,1\}$, and $post_t^i = 1$ if and only if there exists a place $p \in t\bullet$ such that $p^i = 1$. Equation 1 ensures if $post_t^i = 1$ then there exists a place $p \in t\bullet$ such that $p^i = 1$, and Equation 2 ensures if there exists a place $p \in t\bullet$ such that $p^i = 1$ then $post_t^i = 1$.

$$-post_t^i + \sum_{p \in t\bullet} p^i \geq 0 \qquad \forall i \in \{0, \ldots, d\}, \forall t \in T \tag{1}$$

$$p^i - post_t^i \leq 0 \qquad \forall i \in \{0, \ldots, d\}, \forall t \in T, \forall p \in t\bullet \tag{2}$$

We need to specify integer linear equations such that there exists a solution if the following conditions are true:

a $\bullet X_0 \subseteq X_0\bullet$, X_0 is a siphon of N.
b $X_0 \neq \emptyset$, the initial siphon is not empty.
c For all $0 \leq i \leq d$ we have $X_{i+1} \subseteq X_i$, we never add places as we iterate.
d For all $t \in T$ we have $p \in \bullet t$ and $p \in X_{i+1}$ if and only if there exists $p' \in t\bullet$ s.t. $p' \in X_i$.
e For all $p \in X_d$ we have $M_0(p) = 0$, or X_d is not a trap.

The reason we need the second part of condition e is because after d iterations we are not guaranteed to converge on the maximal trap. In order to guarantee convergence, we need the depth to be equal to the number of places, i.e. $d = |P|$. Consider the case where X_d is marked but not a trap and conditions a through d are true. It is then possible that for all places $p \in X_d$ that is marked we have

$p \notin Q_{max}$, which disproves the siphon-trap property. However if we did not have the second part of condition e, this would not be a solution, and if all other siphons contain a marked trap, we incorrectly conclude that the Petri net is deadlock free.

Equation 3 ensures condition a.

$$-p^0 + \sum_{q \in \bullet t} q^0 \geq 0 \qquad \forall t \in T, \forall p \in t\bullet \tag{3}$$

If p is in the initial siphon, i.e. $p^0 = 1$, and it is given a token when t is fired, then we must have at least one place $q^0 = 1$ in the siphon where a token is removed when t is fired, otherwise the equation is not satisfied.

Equation 4 ensures condition b.

$$\sum_{p \in P} p^0 \geq 1 \tag{4}$$

At least one place must be assigned a value of 1 to ensure the initial siphon X_0 is non-empty, otherwise the equation is not satisfied.

Equation 5 ensures condition c.

$$-p^{i+1} + p^i \geq 0 \qquad \forall i \in \{0, \dots, d\}, \forall p \in P \tag{5}$$

If $p^{i+1} = 1$ then we must also have that $p^i = 1$, otherwise the equation is not satisfied. No places can be added in later iterations.

Equation 6 ensures the left-to-right implication of condition d.

$$-p^{i+1} + post_t^i \geq 0 \qquad \forall i \in \{0, \dots, d\}, \forall p \in P, \forall t \in p\bullet \tag{6}$$

Equation 7 ensures the right-to-left implication of condition d.

$$-p^{i+1} + p^i + \sum_{t \in p\bullet} post_t^i \leq |p\bullet| \qquad \forall i \in \{0, \dots, d\}, \forall p \in P \tag{7}$$

We iteratively remove places from the identified siphon until we are either left with the empty set or the maximal trap, iterating d times. A place $p \in X_i$ is removed from the siphon in the ith step by assigning its decision variable p^{i+1} to 0 in step $i+1$, where $p^i = 1$. If place p is not part of the siphon in step i, i.e. $p \notin X_i$ and $p^i = 0$, then it stays outside of the siphon in step $i+1$ and $p^{i+1} = 0$, as we do not add any places. A place p is removed in the ith step if and only if there exists a transition $t \in p\bullet$ s.t. $t\bullet \not\subseteq X_i$.

Once the removal procedure reaches depth d, we are left with one of three cases: Either X_d is the maximal trap, not a trap at all, or the empty set. In either case, we need to check if the set is unmarked. If it is unmarked then X_0 is a siphon with no marked trap, and therefore disproves the siphon-trap property. Let $z \in \mathbb{N}^0$ be a decision variable. Equation 8 ensures the first part of condition e. Equation 9 ensures the second part of condition e.

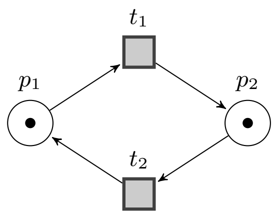


Fig. 5: A Petri net having the siphon-trap property.

$$p^{d+1} - z \leq 0 \qquad\qquad \forall p \in P \text{ where } M_0(p) > 0 \tag{8}$$

$$\sum_{p \in P} p^{d+1} + z = \sum_{p \in P} p^d \tag{9}$$

Now consider the case again where X_d is marked but not the maximal trap. We have that $X_d \subset X_{d-1}$, which forces z to be greater than zero due to equation 9. We know $p^d \in \{0, 1\}$, and because z is not zero we know $p^d - z \leq 0$, and equation 8 is satisfied. So the result is that it is inconclusive whether or not the Petri net have the siphon-trap property, since the chosen depth could not converge on the maximal trap. In the case where X_d is not marked, z can be assigned any value to satisfy equation 8 which makes it trivial to satisfy equation 9.

By the construction and reasoning from the integer linear program specification above, we conclude with the following theorem.

Theorem 2. *If the integer linear program specified in equations 1 through 9 is infeasible then N has no deadlock.*

Example 10 (Siphon-Trap Analysis). Consider the Petri net in Figure 5. We construct an integer linear program and check whether the net has the siphon-trap property. We assign the depth $d = 1$ and generate constraints according to equation 1-9. The following constraints form the integer linear program. We annotate lines with $[n]$, indicating that equation n generates the constraints on

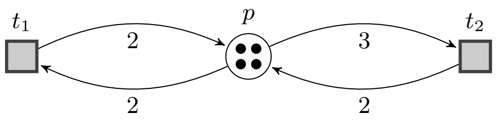


Fig. 6: Example Petri net and initial marking for formula simplification.

that line.

$$\begin{array}{rrr}
-post^0_{t_1} + p^0_2 \geq 0 & -post^0_{t_2} + p^0_1 \geq 0 & [1] \\
p^0_2 - post^0_{t_1} \leq 0 & p^0_1 - post^0_{t_2} \leq 0 & [2] \\
-p^0_2 + p^0_1 \geq 0 & -p^0_1 + p^0_2 \geq 0 & [3] \\
p^0_1 + p^0_2 \geq 1 & & [4] \\
-p^1_1 + p^0_1 \geq 0 & -p^1_2 + p^0_2 \geq 0 & [5] \\
-p^1_1 + post^0_{t_1} \geq 0 & -p^1_2 + post^0_{t_2} \geq 0 & [6] \\
-p^1_1 + p^0_1 + post^0_{t_1} \leq 1 & -p^1_2 + p^0_2 + post^0_{t_2} \leq 1 & [7] \\
p^1_1 - z \leq 0 & p^1_2 - z \leq 0 & [8] \\
p^1_1 + p^1_2 + z = p^0_1 + p^0_2 & & [9]
\end{array}$$

The program consists of 9 decision variables, and there are 16 constraints in total. lp_solve reports that the system of equations is infeasible, meaning that every siphon contains a marked trap, hence the Petri net is deadlock free.

6 Formula Simplification

Previously we have shown how we construct interesting sets of transitions and stubborn sets to verify reachability formulae. Problems can occur with that approach if the formula in question is very large and involves most of the places and transitions in the Petri net. This is because, during verification, we will spend time to recursively traverse the formula at each marking to construct the interesting set of transitions, which after performing the closure ends up including most of the transitions anyway.

For this reason, we need another layer of preprocessing on formulae to reduce the time needed to construct interesting set of transitions. We call this layer *formula simplification*. The general idea of this procedure is that by examining the satisfiability of subformulae, we can replace them with equivalent alternatives that are easier to verify.

To perform formula simplification, we need a way to identify contradictions and impossibilities in the formula. For this, we can use an adaptation of state equations for Petri nets to cardinality formulae, as shown in [11].

Example 11 (State-Equation for Cardinality). Consider the Petri net in Figure 6 with initial marking M_0 where $M_0(p) = 4$, and the reachability formula $EF\ p \geq$

5, i.e. does there exist a reachable marking where the number of tokens in p is five or more. We can model this as an integer linear program, and show that such a marking is unreachable. Let $X = \{x_{t_1}, x_{t_2}\}$ be variables. The proposition $p \geq 5$ can be formulated as a linear equation as follows:

$$M_0(p) + \sum_{t \in T}(W(t,p) - W(p,t))x_t \geq 5 \iff 4 + 0x_{t_1} - 1x_{t_2} \geq 5$$

The weight of x_{t_1} is 0 since, when t_1 is fired, the number of tokens in p remains unchanged. Likewise, the weight of x_{t_2} is -1 because, it removes one token from p after firing. After subtracting 4 from both sides of the inequality, we have the integer linear program with one equation $\{0x_{t_1} - 1x_{t_2} \geq 1\}$. Clearly, there does not exist a non-negative assignment to the variables, such that the linear equation is true. The left hand side can never evaluate to a positive number. There are therefore no reachable marking where the number of tokens in p is more or greater than five, so the formula is not satisfied in the initial marking. We have that $EF\ p \geq 5 \equiv \mathit{false}$ in M_0, and we have simplified the formula to an equivalent but trivial formula.

In [11], the authors construct a set of integer linear programs. They recursively construct this set by traversing the formulae, and after the traversal, they check if there exists an integer linear program in the set that has a solution. If all the integer linear programs do not have a solution, then they can conclude that the formula is not satisfied. We further extend on this in a number of ways.

In [11], comparisons can only have the form $p \bowtie k$, where p is a place and k is a constant. We extend this to allow for the arithmetic expressions as defined in Section 2.3, such as comparing places with each other, addition, subtraction, and multiplication.

The authors in [11] evaluate the set of integer linear programs, only after the formula traversal has terminated. Instead, we evaluate the set of integer linear programs on-the-fly while traversing the formulae, and replace any subformulae with simplified alternatives.

Example 12 (Formula Simplification). As an example, consider the formula $EF\ (p_1 > 5) \lor (p_2 > 2 \land p_2 < 2)$. Assume there exists a reachable marking from the initial marking where $p_1 > 5$ is satisfied, so the formula is satisfied. However, the subformula $p_2 > 2 \land p_2 < 2$ is impossible and equivalent to *false*. With our approach, we are able to identify this impossibility and simplify the formula such that $EF\ (p_1 > 5) \lor (p_2 > 2 \land p_2 < 2) \equiv EF\ p_1 > 5$. Furthermore, we can also handle positive conclusions. Assume that $p_1 > 5$ is always satisfied, i.e. for all reachable markings there are always more than five tokens in p_1. We negate the subformula, $\neg(p > 5) \equiv p \leq 5$, and create an integer linear program for the negated form. If the program is infeasible then it must mean that the formula is always satisfied and equivalent to *true*. So we can conclude that $EF\ (p_1 > 5) \equiv EF\ \mathit{true} \equiv \mathit{true}$.

We extend the technique to also include CTL. Since state equations are tightly knit with reachability, we cannot use the technique directly to decide

CTL formulae. However, we can simplify subformlae to a point where the answer to the CTL formula is trivial, or becomes a reachability formula.

Example 13 (CTL Formula Simplification). As an example, consider the CTL formula $EF\ AX\ \varphi$. Using formula simplification, assume we have determined that $\varphi \equiv \mathit{false}$. The only case where the formula is satisfied is when, in the given marking, there are no enabled transitions available, i.e. $EF\ AX\ \varphi \equiv EF\ \mathit{deadlock}$. So we are able to simplify a CTL formula into a reachability formula.

6.1 Simplification Procedure

We define a function, that given a formula, produces a simplified formula and a set of integer linear programs. We say that such a function is a *simplification* function.

Definition 9 (Simplification). *Let $N = (P, T, W, I)$ be a Petri net, M_0 an initial marking on N, and $X = \{x_t \mid t \in T\}$ a set of variables. A simplification for marking M_0 is a function simplify* $: \Phi_{CTL} \to \Phi_{CTL} \times 2^{\mathcal{E}_{lin}^X}$.

Let $LPS \in 2^{\mathcal{E}_{lin}^X}$ be a set of integer linear programs over a set of variables X. We say that LPS has a solution, if there exists $LP \in LPS$ such that LP has a solution. We write $\mathit{simplify}(\varphi) = (\varphi', LPS)$, when the formula φ has been simplified to φ'. Our goal is to define *simplify* s.t. it holds that $\varphi \equiv \varphi'$ and for any marking $M \in \mathit{reach}(M_0)$ if $M \models \varphi$ then the set of linear programs LPS has a solution.

Let $N = (P, T, W, I)$ be a Petri net. For simplicity, we assume that formulae have been rewritten as seen in Table 5. With these rewriting rules we push negations down to either the atomic propositions on the form $e_1 \bowtie e_2$, *deadlock*, or binary CTL operators $Q(\varphi_1 U \varphi_2)$ where $Q \in \{A, E\}$. In the first case, we replace $\bowtie$ with its dual as seen in Table 2. The second and third case we leave the negations, and, as we will see, are ignored during simplification.

We define a simplification *simplify* on the syntax of CTL formulae for which there are a couple of trivial cases as seen in Table 6.

It is possible to rewrite *deadlock* formulae such that we use conjunction, disjunction, and arithmetic comparisons. We start by rewriting *deadlock* to $\neg t_1 \wedge \neg t_2 \wedge \cdots \wedge \neg t_m$ where $T = \{t_1, t_2, \cdots, t_m\}$, and then rewriting the transition atomic propositions as seen in Table 5 and pushing down negations. We choose not to do this rewriting since the interesting set of transitions for *deadlock* formulae are already small, as seen in Table 1. It is also a very extensive construction which is costly not only during the rewriting but also during the simplification procedure.

The algorithms from Algorithm 2 to Algorithm 10 give the remaining cases of *simplify*.

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 $n = \lvert P \rvert$
$e_1 \neq e_2$	$e_1 > e_2 \vee e_1 < e_2$
$e_1 = e_2$	$e_1 \leq e_2 \wedge e_1 \geq e_2$
$\neg(\varphi_1 \wedge \varphi_2)$	$\neg\varphi_1 \vee \neg\varphi_2$
$\neg(\varphi_1 \vee \varphi_2)$	$\neg\varphi_1 \wedge \neg\varphi_2$
$\varphi_1 \implies \varphi_2$	$\neg\varphi_1 \vee \varphi_2$
$\varphi_1 \iff \varphi_2$	$(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \neg\varphi_2)$
$\neg AX\varphi$	$EX\neg\varphi$
$\neg EX\varphi$	$AX\neg\varphi$
$\neg AF\varphi$	$EG\neg\varphi$
$\neg EF\varphi$	$AG\neg\varphi$
$\neg AG\varphi$	$EF\neg\varphi$
$\neg EG\varphi$	$AF\neg\varphi$

Table 5: Rewriting rules for φ.

φ	$\mathit{simplify}(M_0, \varphi)$
true	$(\mathit{true}, \{\{0 \leq 1\}\})$
false	$(\mathit{false}, \emptyset)$
deadlock	$(\mathit{deadlock}, \{\{0 \leq 1\}\})$

Table 6: Trivial cases of *simplify*.

The function *merge* : $2^{\mathcal{E}^X_{lin}} \times 2^{\mathcal{E}^X_{lin}} \to 2^{\mathcal{E}^X_{lin}}$ combines two LPS and is defined as *merge* $(LPS_1, LPS_2)= \{LP_1 \cup LP_2 \mid LP_1 \in LPS_1$ and $LP_2 \in LPS_2\}$.

Algorithm 2: Simplify $\varphi_1 \wedge \varphi_2$

```
1  Function simplify($\varphi_1 \wedge \varphi_2$)
2      $(\varphi'_1, LPS_1) \leftarrow simplify(\varphi_1)$
3      if $\varphi'_1 = false$ then
4          return $(false, \emptyset)$
5      $(\varphi'_2, LPS_2) \leftarrow simplify(\varphi_2)$
6      if $\varphi'_2 = false$ then
7          return $(false, \emptyset)$
8      else if $\varphi'_2 = true$ then
9          return $(\varphi'_1, LPS_1)$
10     else if $\varphi'_1 = true$ then
11         return $(\varphi'_2, LPS_2)$
12     $LPS \leftarrow merge(LPS_1, LPS_2)$
13     if $\{LP \cup BASE \mid LP \in LPS\}$ has no solution then
14         return $(false, \emptyset)$
15     else
16         return $(\varphi'_1 \wedge \varphi'_2, LPS)$
```

$BASE$ is an integer linear program of a Petri net $N = (P, T, W, I)$ and initial marking M_0 on N, that consists of the following set of linear equations:

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

Which ensures that no solution to the linear program, can leave a place with a negative amount of tokens.

Algorithm 3: Simplify $\varphi_1 \lor \varphi_2$

```
Function simplify($\varphi_1 \lor \varphi_2$)
    $(\varphi_1', LPS_1) \leftarrow simplify(\varphi_1)$
    if $\varphi_1' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    $(\varphi_2', LPS_2) \leftarrow simplify(\varphi_2)$
    if $\varphi_2' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    else if $\varphi_1' = false$ then
        return $(\varphi_2', LPS_2)$
    else if $\varphi_2' = false$ then
        return $(\varphi_1', LPS_1)$
    else
        return $(\varphi_1' \lor \varphi_2', LPS_1 \cup LPS_2)$
```

Algorithm 4: Simplify $\neg\varphi$

```
Function simplify($\neg\varphi$)
    $(\varphi', LPS) \leftarrow simplify(\varphi)$
    if $\varphi' = true$ then
        return $(false, \emptyset)$
    else if $\varphi_2' = false$ then
        return $(true, \{\{0 \leq 1\}\})$
    else
        return $(\neg\varphi', \{\{0 \leq 1\}\})$
```

Merging sets of linear programs ensures that the resulting set of linear programs is feasible, if there exists a union of linear programs, that is feasible. By taking the union, we preserve linear programs with solutions for both sides. If the union of a pair of integer linear programs have no solution, where both had a solution before the union, we can conclude that the sets of reachable markings that satisfy the sides are disjoint.

For the comparison operator $e_1 \bowtie e_2$ we introduce the function *const* which takes as input an expression e and returns one side of a linear equation.

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

Algorithm 5: 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 $(e_1 \bowtie e_2, \{\{0 \leq 1\}\})$
    $LPS_1 \leftarrow \{\{const(e_1) \bowtie const(e_2)\}\}$
    $LPS_2 \leftarrow \{\{const(e_1) \overline{\bowtie} const(e_2)\}\}$
    if $\{LP \cup BASE \mid LP \in LPS_1\}$ have no solution then
        return $(false, \emptyset)$
    else if $\{LP \cup BASE \mid LP \in LPS_2\}$ have no solution then
        return $(true, \{\{0 \leq 1\}\})$
    else
        return $(e_1 \bowtie e_2, LPS_1)$
```

Algorithm 6: Simplify $AX\varphi$

```
Function simplify($AX\varphi$)
    $(\varphi', LPS) \leftarrow simplify(\varphi)$
    if $\varphi' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    else if $\varphi' = false$ then
        return $(deadlock, \{\{0 \leq 1\}\})$
    else
        return $(AX\varphi', \{\{0 \leq 1\}\})$
```

There is a special case we have to handle. If in either of the expressions e_1 or e_2 we have that two places are multiplied together either directly or indirectly. A direct example is $p_1 \cdot p_2$, and an indirect example is $p_1 \cdot (5 + p_2)$. If we multiply places like this, the output of *const* is no longer linear which is a prerequisite for the simplication procedure. To handle this, if either side of the comparison in non-linear, we return the formula unchanged and $\{\{0 \leq 1\}\}$ as the integer linear problem where any variable assignment is a solution. Lastly, we have to prove that the simplification procedure does what is expected of it. In the various cases, we replace subformulae with smaller and easier to verify alternatives whenever the generated set of integer linear programs is infeasible. Correctness of this procedure is to ensure that satisfiability is preserved, which is the focus of Lemma 6. If a simplification satisfies Lemma 6 we call it a *satisfiability preserving simplification.*

Lemma 6 (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. If simplify$(\varphi) = (\varphi', LPS)$ then for all $M \in \mathcal{M}(N)$ such that $M_0 \xrightarrow{w} M$ we have:*

1. *$M \models \varphi$ iff $M \models \varphi'$, and*
2. *if $M \models \varphi$ then there exists $LP \in LPS$ such that $\wp(w)$ is a solution to LP.*

Algorithm 7: Simplify $EX\varphi$

```
Function simplify($EX\varphi$)
    $(\varphi', LPS) \leftarrow simplify(\varphi)$
    if $\varphi' = true$ then
        return $(\neg deadlock, \{\{0 \leq 1\}\})$
    else if $\varphi' = false$ then
        return $(false, \emptyset)$
    else
        return $(EX\varphi', \{\{0 \leq 1\}\})$
```

Algorithm 8: Simplify $QF\varphi$ where $Q \in \{A, E\}$

```
Function simplify($QF\varphi$)
    $(\varphi', LPS) \leftarrow simplify(\varphi)$
    if $\varphi' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    else if $\varphi' = false$ then
        return $(false, \emptyset)$
    else
        return $(QF\varphi', \{\{0 \leq 1\}\})$
```

Proof. The proof proceeds by structural induction on φ. 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. For $simplify(\varphi) = (\varphi', LPS)$ we show for all $M \in \mathcal{M}(N)$ such that $M_0 \xrightarrow{w} M$ that condition 1 and condition 2 holds.

Base Cases:

$\varphi = true$: Since $simplify(true) = (true, \{\{0 \leq 1\}\})$ the formula remains unchanged and condition 1 trivially holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = false$: Since $simplify(false) = (false, \emptyset)$ the formula remains unchanged and condition 1 trivially holds. Condition 2 holds since the premise $M \models \varphi$ never holds.

$\varphi = deadlock$: Since $simplify(deadlock) = (deadlock, \{\{0 \leq 1\}\})$ the formula remains unchanged and condition 1 for formula remains unchanged and trivially holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = e_1 \bowtie e_2$: If either $const(e_1)$ or $const(e_2)$ is not linear, then $simplify(e_1 \bowtie e_2) = (e_1 \bowtie e_2, \{\{0 \leq 1\}\})$. Condition 1 holds since the formula is unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution. Else we have $LPS_1 = \{\{const(e_1) \bowtie const(e_2)\}\}$ and $LPS_2 = \{\{const(e_1) \overline{\bowtie} const(e_2)\}\}$. There are 3 cases:

1. $simplify(\varphi) = (false, \emptyset)$: If LPS_1 have no solution then for all $M' \in reach(M_0)$ we have that $M' \not\models e_1 \bowtie e_2$, and we know $M \in reach(M_0)$.

Algorithm 9: Simplify $QG\varphi$ where $Q \in \{A, E\}$

```
Function $simplify(QG\varphi)$
    $(\varphi', LPS) \leftarrow simplify(\varphi)$
    if $\varphi' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    else if $\varphi' = false$ then
        return $(false, \emptyset)$
    else
        return $(QG\varphi', \{\{0 \leq 1\}\})$
```

Algorithm 10: Simplify $Q(\varphi_1 U \varphi_2)$ where $Q \in \{A, E\}$

```
Function $simplify(Q(\varphi_1 U \varphi_2))$
    $(\varphi_2', LPS_2) \leftarrow simplify(\varphi_2)$
    if $\varphi_2' = true$ then
        return $(true, \{\{0 \leq 1\}\})$
    else if $\varphi_2' = false$ then
        return $(false, \emptyset)$
    $(\varphi_1', LPS_1) \leftarrow simplify(\varphi_1)$
    if $\varphi_1' = true$ then
        return $(QF\varphi_2', \{\{0 \leq 1\}\})$
    else if $\varphi_1' = false$ then
        return $(\varphi_2', LPS_2)$
    else
        return $(Q(\varphi_1' U \varphi_2'), \{\{0 \leq 1\}\})$
```

We therefore have that $M \not\models e_1 \bowtie e_2$ and $e_1 \bowtie e_2 \equiv false$ in M, so condition 1 holds. Condition 2 is trivial since the premise $M \models \varphi$ never holds.

2. $simplify(\varphi) = (true, \{0 \leq 1\})$: If LPS_2 have no solution then for all $M' \in reach(M)$ we have that $M' \not\models e_1 \overline{\bowtie} e_2$, which is equivalent to $M' \models e_1 \bowtie e_2$. Since $M \in reach(M_0)$ we have that $M \models e_1 \bowtie e_2$ and $e_1 \bowtie e_2 \equiv true$ in M, so condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
3. $(e_1 \bowtie e_2, LPS_1)$: The formula is unchanged so condition 1 holds. Assuming $M \models e_1 \bowtie e_2$, then since $M \in reach(M_0)$ we know that $\wp(w)$ is a solution to the integer linear program $const(e_1) \bowtie const(e_2)$, and condition 2 holds.

Inductive Cases:

$\varphi = \varphi_1 \vee \varphi_2$: Let $simplify(\varphi_1) = (\varphi_1', LPS_1)$, $simplify(\varphi_2) = (\varphi_2', LPS_2)$, and $M_0 \xrightarrow{w} M$. By structural induction, $\varphi_1 \equiv \varphi_1'$ and $\varphi_2 \equiv \varphi_2'$.

If $\varphi_1' = \mathit{true}$ or $\varphi_2' = \mathit{true}$, then $\varphi_1 \vee \varphi_2 \equiv \mathit{true}$, $\mathit{simplify}(\varphi_1 \vee \varphi_2) = (\mathit{true}, \{\{0 \leq 1\}\})$, and condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

If $\varphi_1' = \mathit{false}$, then $\varphi_1 \vee \varphi_2 \equiv \varphi_2'$, $\mathit{simplify}(\varphi_1 \vee \varphi_2) = (\varphi_2', \mathit{LPS}_2)$, and condition 1 holds. If $M \models \varphi$ then we have that $M \models \varphi_2$ and by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_2$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds.

If $\varphi_2' = \mathit{false}$, then $\varphi_1 \vee \varphi_2 \equiv \varphi_1'$, $\mathit{simplify}(\varphi_1 \vee \varphi_2) = (\varphi_1', \mathit{LPS}_1)$, and condition 1 holds. If $M \models \varphi$ then we have that $M \models \varphi_1$ and by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_1$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds.

If φ_1' and φ_2' are not *true* or *false*, then $\varphi_1 \vee \varphi_2 \equiv \varphi_1' \vee \varphi_2'$, $\mathit{simplify}(\varphi_1 \vee \varphi_2) = (\varphi_1' \vee \varphi_2', \mathit{LPS}_1 \cup \mathit{LPS}_2)$, and condition 1 holds. If $M \models \varphi$ then either $M \models \varphi_1$ or $M \models \varphi_2$. If $M \models \varphi_1$ then by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_1$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds since $\mathit{LP} \in \mathit{LPS}_1 \cup \mathit{LPS}_2$. If $M \models \varphi_2$ then by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_2$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds since $\mathit{LP} \in \mathit{LPS}_1 \cup \mathit{LPS}_2$.

$\varphi = \varphi_1 \wedge \varphi_2$: Let $\mathit{simplify}(\varphi_1) = (\varphi_1', \mathit{LPS}_1)$, $\mathit{simplify}(\varphi_2) = (\varphi_2', \mathit{LPS}_2)$, and $M_0 \xrightarrow{w} M$. By structural induction, $\varphi_1 \equiv \varphi_1'$, $\varphi_2 \equiv \varphi_2'$, and $\varphi_1 \wedge \varphi_2 \equiv \varphi_1' \wedge \varphi_2'$.

If $\varphi_1' = \mathit{false}$ or $\varphi_2' = \mathit{false}$, then $\varphi_1 \wedge \varphi_2 \equiv \mathit{false}$, $\mathit{simplify}(\varphi_1 \wedge \varphi_2) = (\mathit{false}, \emptyset)$, and condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.

If $\varphi_1' = \mathit{true}$, then $\varphi_1 \wedge \varphi_2 \equiv \varphi_2'$, $\mathit{simplify}(\varphi_1 \wedge \varphi_2) = (\varphi_2', \mathit{LPS}_2)$, and condition 1 holds. If $M \models \varphi$ then we have that $M \models \varphi_2$ and by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_2$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds.

If $\varphi_2' = \mathit{true}$, then $\varphi_1 \wedge \varphi_2 \equiv \varphi_1'$, $\mathit{simplify}(\varphi_1 \wedge \varphi_2) = (\varphi_1', \mathit{LPS}_1)$, and condition 1 holds. If $M \models \varphi$ then we have that $M \models \varphi_1$ and by the induction hypothesis there exists $\mathit{LP} \in \mathit{LPS}_1$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds.

If φ_1' and φ_2' are not *true* or *false*, then $\varphi_1 \wedge \varphi_2 \equiv \varphi_1' \wedge \varphi_2'$, $\mathit{simplify}(\varphi_1 \wedge \varphi_2) = (\varphi_1' \wedge \varphi_2', \mathit{merge}(\mathit{LPS}_1, \mathit{LPS}_2))$, and condition 1 holds. If $M \models \varphi$ then $M \models \varphi_1$ and $M \models \varphi_2$ and by the induction hypothesis there exists $\mathit{LP}_1 \in \mathit{LPS}_1$ and $\mathit{LP}_2 \in \mathit{LPS}_2$ s.t. $\wp(w)$ is a solution to LP_1 and LP_2. We know $\mathit{LP}_1 \cup \mathit{LP}_2 \in \mathit{merge}(\mathit{LPS}_1, \mathit{LPS}_2)$ by the definition of *merge*. If $\wp(w)$ is not a solution to $\mathit{LP}_1 \cup \mathit{LP}_2$ then it contradicts that M satisfies both φ_1 and φ_2, so $\wp(w)$ has to be a solution to $\mathit{LP}_1 \cup \mathit{LP}_2$ and condition 2 holds.

$\varphi = \neg\varphi_1$: Let $\mathit{simplify}(\varphi_1) = (\varphi_2, \mathit{LPS})$. By structural induction we have that $\varphi_1 \equiv \varphi_2$.

If $\varphi_2 = \mathit{false}$ then we have that $\neg\varphi_1 \equiv \mathit{true}$, $\mathit{simplify}(\varphi_1) = (\mathit{true}, \{\{0 \leq 1\}\})$, and condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

If $\varphi_2 = \mathit{true}$ then we have that $\neg\varphi_1 \equiv \mathit{false}$, $\mathit{simplify}(\varphi_1) = (\mathit{false}, \emptyset)$, and condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.

Else $\mathit{simplify}(\neg\varphi_1) = (\neg\varphi_2, \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = AX\varphi_1$: Let $\mathit{simplify}(\varphi)_1 = (\varphi_2, \mathit{LPS})$. By structural induction we have that $\varphi_1 \equiv \varphi_2$.
If $\varphi_2 = \mathit{true}$ then we have that $AX\varphi_1 \equiv \mathit{true}$ and $\mathit{simplify}(\varphi) = (\mathit{true}, \{\{0 \leq 1\}\})$. This is because for all $M' \in \mathcal{M}(N)$ if $M \to M'$ we have $M' \models \mathit{true}$ is trivially true. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
If $\varphi_2 = \mathit{false}$ then we have that $AX\varphi_1 \equiv \mathit{deadlock}$ and $\mathit{simplify}(\varphi) = (\mathit{deadlock}, \{\{0 \leq 1\}\})$. This is because the only case where $M \models AX\mathit{false}$ is when $en(M) = \emptyset$, i.e. M is a deadlock. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
Else $\mathit{simplify}(AX\varphi_1) = (AX\varphi_2, \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = EX\varphi_1$: Let $\mathit{simplify}(\varphi_1) = (\varphi_2, \mathit{LPS})$. By structural induction we have that $\varphi_1 \equiv \varphi_2$.
If $\varphi_2 = \mathit{true}$ then we have that $AX\varphi_1 \equiv \neg\mathit{deadlock}$ and $\mathit{simplify}(\varphi) = (\neg\mathit{deadlock}, \{\{0 \leq 1\}\})$. This is because the only case where $M \models EX\mathit{true}$ is when $en(M) \neq \emptyset$, i.e. M is not a deadlock. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
If $\varphi_2 = \mathit{false}$ then we have that $EX\varphi_1 \equiv \mathit{false}$ and $\mathit{simplify}(\varphi) = (\mathit{false}, \emptyset)$. This is because if there exists $M' \in \mathcal{M}(N)$ s.t. $M \to M'$ then $M \models \mathit{false}$ is trivially false. Therefore condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.
Else $\mathit{simplify}(EX\varphi_1) = (EX\varphi_2, \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = EF\varphi_1$: Let $\mathit{simplify}(\varphi_1) = (\varphi_2, \mathit{LPS})$. By structural induction we have that $\varphi_1 \equiv \varphi_2$.
If $\varphi_2 = \mathit{true}$ then we have that $EF\varphi_1 \equiv \mathit{true}$ and $\mathit{simplify}(\varphi) = (\mathit{true}, \{\{0 \leq 1\}\})$, since $EF\mathit{true}$ is trivially true. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
If $\varphi_2 = \mathit{false}$ then we have that $EF\varphi_1 \equiv \mathit{false}$ and $\mathit{simplify}(\varphi) = (\mathit{false}, \emptyset)$, since $EF\mathit{false}$ is trivially false. Therefore condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.
Else $\mathit{simplify}(EF\varphi_1) = (EF\varphi_2, \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = AF\varphi_1$: The same as the $EF\varphi_1$ case.

$\varphi = EG\varphi_1$: Let $\mathit{simplify}(\varphi_1) = (\varphi_2, \mathit{LPS})$. By structural induction we have that $\varphi_1 \equiv \varphi_2$.
If $\varphi_2 = \mathit{true}$ then we have that $EG\varphi_1 \equiv \mathit{true}$ and $\mathit{simplify}(\varphi) = (\mathit{true}, \{\{0 \leq 1\}\})$, since $EF\mathit{true}$ is trivially true. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

If $\varphi_2 = \mathit{false}$ then we have that $EG\varphi_1 \equiv \mathit{false}$ and $\mathit{simplify}(\varphi) = (\mathit{false}, \emptyset)$, since $EG\mathit{false}$ is trivially false. Therefore condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.
Else $\mathit{simplify}(EG\varphi_1) = (EG\varphi_2, \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = AG\varphi_1$: The same as the $EG\varphi_1$ case.

$\varphi = E(\varphi_1 U \varphi_2)$: Let $\mathit{simplify}(\varphi_1) = (\varphi_1', \mathit{LPS}_1)$ and $\mathit{simplify}(\varphi_2) = (\varphi_2', \mathit{LPS}_2)$. By structural induction we have that $\varphi_1 \equiv \varphi_1'$ and $\varphi_2 \equiv \varphi_2'$.
If $\varphi_2' = \mathit{true}$ then we have that $E(\varphi_1 U \varphi_2) \equiv \mathit{true}$ and $\mathit{simplify}(\varphi) = (\mathit{true}, \{\{0 \leq 1\}\})$. This is because for all $M' \in \mathit{reach}(M)$ we have $M' \models \mathit{true}$ is trivially true. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
If $\varphi_2' = \mathit{false}$ then we have that $E(\varphi_1 U \varphi_2) \equiv \mathit{false}$ and $\mathit{simplify}(\varphi) = (\mathit{false}, \emptyset)$. This is because for all $M' \in \mathit{reach}(M)$ we have $M' \models \mathit{false}$ is trivially false. Therefore condition 1 holds. Condition 2 holds since the premise $M \models \varphi$ never holds.
If $\varphi_1' = \mathit{true}$ then we have that $E(\varphi_1 U \varphi_2) \equiv EF\varphi_2'$ and $\mathit{simplify}(\varphi) = (EF\varphi_2', \{\{0 \leq 1\})$. Therefore condition 1 holds. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.
If $\varphi_1' = \mathit{false}$ then we have that $E(\varphi_1 U \varphi_2) \equiv \varphi_2'$ and $\mathit{simplify}(\varphi) = (\varphi_2', \mathit{LPS}_2)$. This is because if $M \not\models \varphi_2'$ then we need to find an $M' \in \mathit{reach}(M)$ where $M' \models \varphi_2'$ and every intermediary marking from M to M' satisfies false, which is never the case. So the only case where the formula is satisfied is when $M \models \varphi_2'$. Therefore condition 1 holds. If $M \models \varphi$ then we have that $M \models \varphi_2$ and by the induction hypothesis there exists $LP \in \mathit{LPS}_2$ s.t. $\wp(w)$ is a solution to LP, and condition 2 holds.
Else $\mathit{simplify}(\varphi) = (E(\varphi_1' U \varphi_2'), \{\{0 \leq 1\}\})$ and condition 1 holds because the formula remains unchanged. Condition 2 holds because for $\{0 \leq 1\}$ any variable assignment is a solution.

$\varphi = A(\varphi_1 U \varphi_2)$: The same as the $E(\varphi_1 U \varphi_2)$ case.

□

7 Implementation

The techniques described in the previous sections, formula simplification, siphon-trap analysis, and stubborn reduction, have all been integrated into the TAPAAL toolchain. The techniques are implemented in C++, and we utilise lp_solve [3], an integer linear program solver, in both formula simplification and siphon-trap analysis. We utilise RapidXML [14] to effectively parse the XML input files. The implementation and experiment data is available at launchpad in the branch `https://code.launchpad.net/~verifypn-stub/verifypn/masters-thesis`.

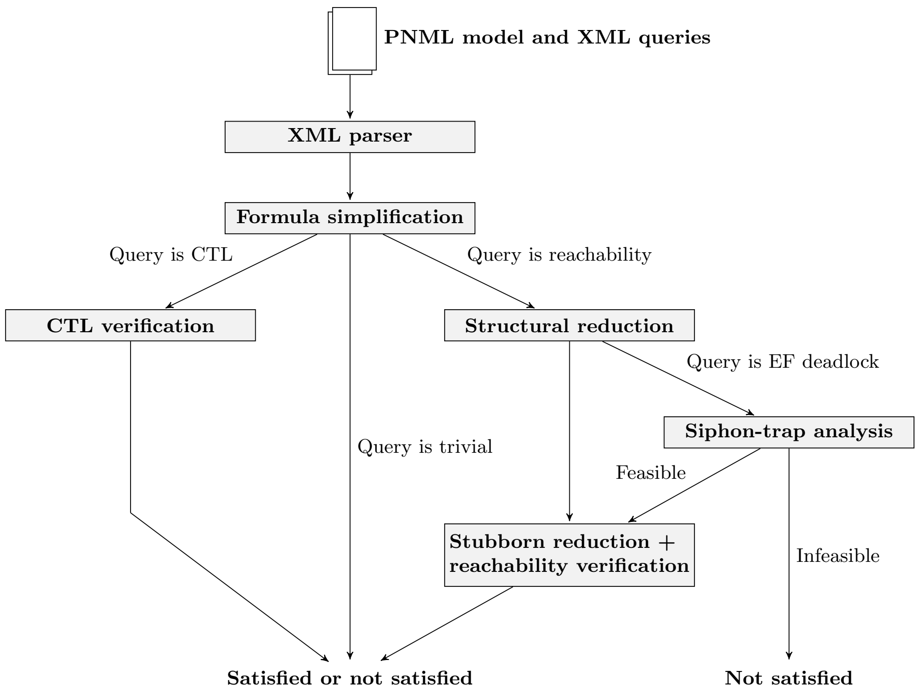


Fig. 7: The TAPAAL toolchain and data flow.

7.1 Toolchain

TAPAAL takes as input a Petri net and a query, performs an analysis, and then determines the satisfiability of the query. The toolchain consists of the following components:

– XML Parser
– Formula simplification
– Structural reduction
– Siphon-trap analysis
– Stubborn reduction verification
– CTL verification

The different components attempts to reduce or modify the structure of the input, which possibly results in an early termination. If the analysis does not terminate, then the modified input is passed to the subsequent component. The components are connected as illustrated in Figure 7.

Both the Petri net and query are initially parsed and stored internally. We have extended the XML parser to handle CTL queries. We transform fireability queries to cardinality queries in the parser.

A description of the data structure used for storing the state space is found in [13]. Experiments show that the representation reduces the required memory for storing the net, and improves the successor generation when performing state space analysis. We build our stubborn reduction on top of this representation.

Queries are represented using a tree-like class structure, where Boolean expression inherits from the *Condition* class. Some of the implemented member functions are shown in the class definition in listing 1.1.

```
class Condition {
public:
    /** Query Simplification */
    virtual Retval simplify(SimplificationContext& context) const = 0;
    /** Check if query is a reachability query */
    virtual bool isReachability(uint32_t depth = 0) const = 0;
    /** Evaluate condition */
    virtual bool evaluate(const EvaluationContext& context) const = 0;
    /** Find interesting transitions in stubborn reduction */
    virtual void findInteresting(
      ReducingSuccessorGenerator& generator,
      bool negated) const = 0;
```

Listing 1.1: Condition class member functions.

Information is passed between objects using different context classes, and the *Retval* class is used as the return type for the simplification methods, which contains a set of linear programs and a formula.

After the net and query have been parsed, we attempt to simplify the query. Section 6 describes this procedure, which results in either a conclusive answer, or a possibly simplified query. If the simplified query is a reachability query, we can perform structural reduction. The simplification process can take a long time to finish, especially due to the processing of conjunction operators. Therefore, we have added a controllable timeout, that terminates the simplification, but without destroying the already simplified part of the query. Listing 1.2 shows the simplification function for conjunction operators.

When *simplifyAnd* is called, then both children of the conjunction operator have already been simplified, and the result stored in *r1* and *r2*. In line 4-10 we evaluate whether either of the children is a true or false formula, and simplify the formula accordingly. We proceed if the formula was not simplified. In line 12 we check whether the process has exceeded the timeout limit, and clears the linear programs before returning from the function. If the timeout was not exceeded, then we merge the two linear programs, and check whether one of the resulting linear programs is infeasible.

We utilise the preprocess functionality *presolve* of lp_solve. Presolve attempts to reduce the the given integer linear program, for instance by eliminating linearly dependent constraints, or deleting unused variables and constraints.

The simplification logic is wrapped in a try block, where we catch out-of-memory exceptions. If the simplification runs out of memory, usually due to the *merge* function in line 13, then *simplifyAnd* returns the smallest of *r1* and *r2*. This ensures that at least one of the linear programs is preserved, thereby increasing the possibility of having contradicting constraints, that makes one of the linear programs infeasible.

```
Retval simplifyAnd(SimplificationContext& context,
  Retval&& r1, Retval&& r2) {
try{
  if(r1.formula->isTriviallyFalse() || r2.formula->isTriviallyFalse()) {
      return Retval(BooleanCondition::FALSE);
  } else if (r1.formula->isTriviallyTrue()) {
      return std::move(r2);
  } else if (r2.formula->isTriviallyTrue()) {
      return std::move(r1);
  }

  if(!context.timeout()) {
      r1.lps.merge(r2.lps);
      if(!context.timeout() && !r1.lps.satisfiable(context)) {
          return Retval(BooleanCondition::FALSE);
      }
  } else {
      r1.lps.clear();
      r2.lps.clear();
  }
  return Retval(std::make_shared<AndCondition>(r1.formula, r2.formula),
    std::move(r1.lps));
} catch(std::bad_alloc& e) {
  // We are out of memory
  return Retval(std::make_shared<AndCondition>(r1.formula, r2.formula),
    std::move((r1.lps.size() < r2.lps.size() ? r1.lps : r2.lps)));
}}
```

Listing 1.2: Simplification function for conjunction operators.

```
void LessThanCondition::findInteresting(
        ReducingSuccessorGenerator& generator, bool negated) const {
    if (!negated) {                 // less than
        if (_expr1->getEval() < _expr2->getEval()) { return; }
        _expr1->decr(generator);
        _expr2->incr(generator);
    } else {                        // greater than or equal
        if (_expr1->getEval() >= _expr2->getEval()) { return; }
        _expr1->incr(generator);
        _expr2->decr(generator);
    }
}
```

Listing 1.3: Function for finding interesting transitions for less-than operators.

We measure the size of the query by counting the number of Boolean and arithmetic expressions, to be able to output statistics on how effective the reduction is. If the satisfiability of the query cannot be answered trivially, then the query and net are passed to the structural reduction component. The structural reduction rules are described in [11].

At this point we detect the type of the query, and there are three scenarios on how the verification proceeds. If the query is simplified to *EF deadlock* then we perform siphon-trap analysis. If no solution was found, then we continue with regular verification using state space exploration. If the query is a CTL query, then it is passed to the CTL verification engine, and if the query is simplified to a reachability query, then we verify the query using stubborn reduction.

```
Size of net before structural reduction: 26 places, 18 transitions
Size of net after structural reduction: 5 places, 4 transitions
Structural reduction finished after 0.000279 seconds

Net reduction is enabled.
Removed transitions: 14
Removed places: 21
Applications of rule A: 10
Applications of rule B: 4
Applications of rule C: 7
Applications of rule D: 0
Applications of rule E: 0

Query is NOT satisfied.

STATS:
        discovered states: 5
        explored states:   5
        expanded states:   5
        max tokens:        4
```

Listing 1.4: Reachability statistics example output.

In Table 3 we show how the set of interesting transitions are found with respect to the comparison operators in the CTL logic. The implementation of the less than operator is shown in listing 1.3.

In line 3 we detect whether we have discovered a negation operator earlier in the query. If no negation was found, then we treat the object as a 'less than' operator, otherwise we treat it as its complement operator 'greater than or equal'. The first check in each case, in line 4 and 8, is to evaluate whether the formula rooted from the current object is satisfied. This is because we only consider those transitions that can alter the truth value of the formula from *false* to *true*, as explained in Section 4. We then call the functions *decr* and *incr* that find the interesting transitions based on the discovered places in the query.

When the reachability or CTL verification has terminated, we output statistics and provide an answer to the satisfiability of the query. An example output from a verification of the model *HouseConstruction-PT-002* is shown in listing 1.4.

Example 14. Consider the Petri net in Figure 6, and the CTL formula $E\ (p \geq 5\ U\ EF\ p = 2)$. Before simplifying the 'until' operator, we first evaluate $\mathit{simplify}(p \geq 5)$. By extension of Example 11, the sub formulae $p \geq 5$ is trivially *false*. Now we have $E\ (\mathit{false}\ U\ EF\ p = 2)$. We cannot simplify $EF\ p = 2$ any further than what it is in this particular case, as $\mathit{simplify}(EF\ p = 2) = (EF\ p = 2, LPS_2)$, where $LPS_2 = \{\{0x_{t_1} - 1x_{t_2} = -2\}\}$ which clearly has a solution in $x_{t_2} = 2$. Algorithm 10 states that in this scenario where the first child of an 'until' is *false*, we return $(EF\ p = 2, LPS_2)$. We have now simplified a CTL formula to a reachability formula, and this allows us utilise the efficient data structures and algorithms of the reachability verification engine of TAPAAL, such as stubborn reduction.

8 Experiments

We evaluate the performance of our implementation and compare it to state of the art model checker LoLA. The implemented techniques are siphon-trap analysis, stubborn reduction, and formula simplification, and we measure the verification performance on the Petri net model database from MCC'17.

8.1 Tools and Configurations

All experiments are performed using the TAPAAL untimed engine[1] [11] with our implementation additions, and LoLA[2] [28]. We explore various configurations of both tools. Table 7 shows an overview of the tool configurations used throughout the experiments.

8.1.1 TAPAAL configurations

For formula simplification, TAPAAL uses the techniques described in Section 6, and is applied to all query categories. The structural reduction technique used in TAPAAL, applies a set of rules to remove places and transitions while preserving certain properties in the reduced net, including cardinality properties [11]. For siphon-trap analysis, TAPAAL translates the siphon-trap property into an integer linear program, as the formula described in Section 5, and utilises lp_solve [3] to solve it. The stubborn reduction technique applies the closure method presented in Algorithm 1, and is applied to reachability analysis only. An older version of TAPAAL [6] is used to perform state equations, that does linear-algebraic over-approximation [11].

8.1.2 LoLA configurations

The chosen configurations of LoLA are based on an email correspondence with Torsten Liebke of the LoLA team. To do stubborn reduction, LoLA uses Tarjan's algorithm [23] that identifies strongly connected components. Stubborn reduction is applied to reachability analysis only. For state equations, LoLA uses counterexample guided abstraction refinement (CEGAR) [27] which can be performed in parallel with state space exploration. The siphon-trap analysis technique in LoLA is done by translating the siphon-trap property into a Boolean satisfiability problem, and utilising MiniSAT [7] to solve it. The siphon-trap analysis can be performed in parallel with state space exploration. If possible, LoLA performs several query transformations [1], similar to our formula simplification. LoLA always performs formula simplification, and hence all LoLA configurations of Table 7 are implicitly using formula simplification.

[1] verifypn 2.1.0: `https://code.launchpad.net/~verifypn-maintainers/verifypn/new-trunk`

[2] LoLA development: `http://svn.gna.org/svn/service-tech/trunk/lola2/`, check out March 16.

Abbreviation	Tool configuration
Base	TAPAAL using only exhaustive search
SE	TAPAAL 2.0 using state equations without state space exploration
Stub	TAPAAL using stubborn reduction
Struct	TAPAAL using structural reduction
StubStruct	TAPAAL using stubborn and structural reduction
Simp	TAPAAL using formula simplification
SimpOnly	TAPAAL using formula simplification without state space exploration
SimpStub	TAPAAL using formula simplification and stubborn reduction
SimpStruct	TAPAAL using formula simplification and structural reduction
Siphon	TAPAAL using siphon-trap analysis without state space exploration
StructSiphon	TAPAAL using structural reduction and siphon-trap analysis without state space exploration
Best	TAPAAL using stubborn reduction, structural reduction, formula simplification, and siphon-trap analysis
LSiphon	LoLA using siphon-trap analysis without state space exploration
LoLA	LoLA using stubborn reduction, running state equations in parallel, and running siphon-trap analysis in parallel

Table 7: Configurations of TAPAAL and LoLA as used in the experiments.

8.2 Model database and experiment metrics

All experiments are performed using the known Petri net models, from the MCC'17 model database. The model database consists of 313 non-colored Petri net model instances from both academic and industrial cases. The models do not contain any inhibitor arcs. Each model is associated with 16 queries from different categories of logic. We perform experiments on the initial 5 queries from the following categories: ReachabilityCardinality, ReachabilityFireability, CTLCardinality, CTLFireability, and the single query of ReachabilityDeadlock.

We do pairwise comparison of configurations. For each query, a configuration is given a point relative to another configuration, whenever it:

exclusive: Answers the query exclusively.

time: Answers the query at least 10% faster, disregarding queries that are solved in less than 10 seconds by both configurations.

states: Answers the query by exploring fewer states, disregarding queries that are exclusively answered.

memory: Answers the query by using less peak memory, disregarding queries that are exclusively answered.

We measure the difficulty of queries by the time used to verify them, by a given configuration. When presenting the most difficult instances, we refrain from showing more than one kind of a model in a given category to ensure the presented results are diverse.

			search strategy	
category (abbreviation)	**queries**	**timeout**	**TAPAAL**	**LoLA**
Reachability cardinality (**RC**)	1565	15 minutes	Heuristic	DFS
Reachability fireability (**RF**)				
CTL cardinality (**CC**)			DFS	
CTL fireability (**CF**)				
Reachability deadlock (**DL**)	313	1 hour		

Table 8: Overview of experiment setup, including category abbreviations, total number of queries per category, timeout per query, and search strategy used for each category for configurations of TAPAAL and LoLA.

8.3 Setup

All experiments were performed on a cluster of 9 compute nodes, each having 1 TB of memory, four AMD Opteron 6376 CPU's, and running Ubuntu 14.04. We run 8 processes per compute node simultaneously, which can cause slight variance in computation time of $\sim 5\%$.

Table 8 shows the query category details and verification setup. From each of the cardinality and fireability categories, we run experiments on 1565 queries for 15 minutes per query. From the reachability deadlock category, we run experiments on 313 queries for 1 hour per query. All configurations have 100GB available memory per query. We use this setup in each experiment, unless otherwise is explicitly stated.

8.4 Combining stubborn reduction and structural reduction

We perform experiments on queries from the reachability categories (**RC**, **RF**, **DL**) to examine the performance of stubborn reduction. We are interested in comparing and combining the technique with structural reduction. We expect that combining the reduction techniques will yield the best state space reduction in general. We compare four tool configurations: **Base** as reference for plain exhaustive state space search, **Stub** that applies stubborn reduction, **Struct** with structural reduction, and **StubStruct** that combines stubborn and structural reduction.

Table 9 shows the number of queries solved by each configuration. We see that stubborn reduction generally provides more answers, and the combination of stubborn reduction and structural reduction provides the most answers in all three categories.

Pairwise score comparison between the four configurations is shown in Table 10. Stubborn reduction compared to no reduction in Table 10a shows a significant improvement across all score metrics. We expect the points in favor of **Base** is due to models not being very reducible by stubborn reduction, causing **Stub** to give unnecessary overhead. The search strategy can be changed by the stubborn reduction, which can also cause the tool to search in a different and longer state space path. Stubborn reduction also performs overall better than structural reduction in Table 10d, however, there is a considerable number queries where structural reduction answers exclusively or simply more efficient in terms of time, states and memory. When comparing Table 10a, 10b and 10c, we see the the combined efficiency of stubborn and structural reduction. There is a significant increase in the number of exclusive answers when combining the reductions. We notice that, when adding stubborn reduction to structural reduction, there appears a few new exclusive answers to **Base** and a considerable number of faster answers. This suggests that the reduction techniques conflict on a number of instances and the stubborn reduction loses efficiency after structural reduction is applied and we are left with the computational overhead. Table 10f displays the contribution of adding stubborn reduction to TAPAAL. Structural reduction is the current best state space exploration technique in the tool, and the addition of stubborn reduction increases performance significantly.

In Table 11 we highlight the most difficult instances and how the four configurations perform on them. On instance **M1** we see an example of both structural reduction and stubborn reduction performing better than plain state space search, and in combination they perform even better in time, memory and states explored. On multiple instances, the stubborn reduction do not affect the number of explored states at all. On instance **M9** the stubborn reduction worsens performance.

To examine the extension of inhibitor arcs to stubborn reduction, we reconstruct a model from the MMC'17 databse, SwimmingPool, with included inhibitor arcs, illustrated in Figure 8. We scale the model by number of tokens in the initial marking and weights on all inhibitor arcs to measure the scalability of our stubborn reduction implementation. We perform deadlock analysis on the model instances and measure the time and explored states in Table 12. The model does not contain deadlocks, so the configurations must explore all states needed to verify the absence of deadlocks. For **Base** this means exploring the complete state space, and for **Stub** it means exploring a possibly reduced state space. We see in Table 12 that the stubborn reduction significantly reduces the state space and scales overall better.

		number of queries solved			
cat.	**queries**	**Base**	**Stub**	**Struct**	**StubStruct**
RC	1565	817	961	879	996
RF	1565	1082	1184	1125	1212
DL	313	211	238	221	239
total	3443	2110	2383	2225	2447

Table 9: Number of queries solved by each configuration.

Base vs Stub								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	13	157	80	208	22	401	9	144
RF	9	111	71	176	22	584	10	184
DL	9	152	79	204	28	482	10	157
total	31	420	230	588	72	1467	29	485

(a)

Base vs Struct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	2	64	31	97	17	257	4	75
RF	2	45	40	89	16	306	4	95
DL	0	10	5	19	17	79	0	23
total	4	119	76	205	50	642	8	193

(b)

Base vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	11	190	90	245	22	477	9	155
RF	9	139	87	213	21	667	8	196
DL	2	30	10	47	23	154	4	45
total	22	359	187	505	66	1298	21	396

(c)

Stub vs Struct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	115	33	183	104	337	186	124	32
RF	87	28	169	94	512	186	154	37
DL	20	3	37	9	131	47	32	7
total	222	64	389	207	980	419	310	76

(d)

Stub vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	2	37	37	84	14	320	5	74
RF	6	34	50	69	23	326	6	61
DL	0	1	5	10	15	88	0	12
total	8	72	92	163	52	734	11	147

(e)

Struct vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	22	139	88	192	20	429	9	141
RF	12	99	73	156	22	589	8	158
DL	2	20	9	35	14	132	4	33
total	36	258	170	383	56	1150	21	332

(f)

Table 10: Pairwise score comparisons. In categories **RC** and **RF** there are 1565 queries, and **DL** contains 313 queries.

	Base			**Stub**			**Struct**			**StubStruct**		
model	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**
M1	> 900	1.99		197	609	19689	887	1612	59408	56	183	6145
M2	853	4609	124235	235	1743	35337	825	4608	124235	233	1743	35337
M3	769	1424	280	724	1426	280	746	1426	280	770	1426	280
M4	> 900	2		863	7182	2242	35	200	10.7	35	200	0.04
M5	756	551	3.24	759	551	3.24	800	552	3.43	829	551	3.43
M6	695	150	36.6	690	151	36.6	702	149	36.6	700	149	36.6
M7	> 3600	1.88		0	3.31	1.01	3493	14429	551952	0	3.2	0.3
M8	3318	3683	131128	0	3.71	15.3	3373	3680	131128	0	3.71	15.3
M9	784	186	5145	2066	910	25386	347	194	5145	1101	928	25402

Table 11: The most difficult instances for at least one of **Stub**, **Struct**, and **StubStruct**, where all three configurations provide answer. Time is in seconds, memory is in MB, states is in number of thousands. Models (name, category, query index): **M1**: PermAdmissibility-PT-05, **RC**, 4. **M2**: CloudDeployment-PT-2b, **RC**, 4. **M3**: HouseConstruction-PT-200, **RC**, 3. **M4**: GlobalResAllocation-PT-05, **RF**, 2. **M5**: DatabaseWithMutex-PT-40, **RF**, 2. **M6**: IBMB2S565S3960-PT-none, **RF**, 5. **M7**: Diffusion2D-PT-D05N010, **DL**, 1. **M8**: CloudDeployment-PT-5a, **DL**, 1. **M9**: GPPP-PT-C0001N0000000100, **DL**, 1.

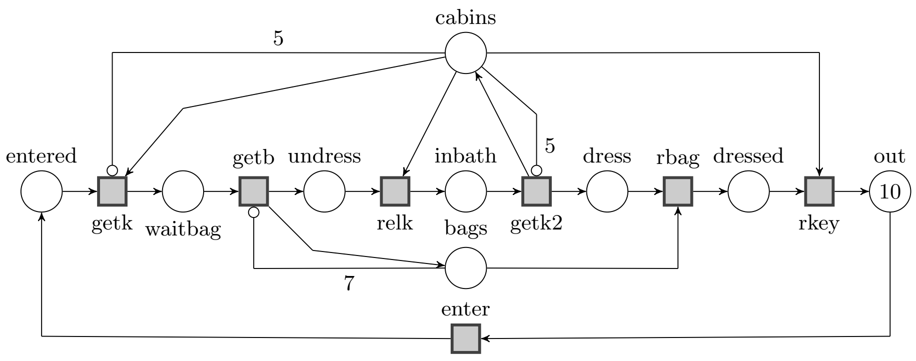


Fig. 8: Model 'SwimmingPool' with inhibitor arcs. Place 'out' has 10 tokens in the initial marking.

	Base		**Stub**	
size	**time**	**states**	**time**	**states**
1	0.08	3366	0.04	740
2	0.29	87297	0.07	7985
3	2.45	707196	0.11	35144
4	13.33	3320086	0.31	103337
5	52.56	11333790	0.71	241397
6	157.97	31381878	1.44	485870
7	413.54	74924382	2.5	881015
8	936.53	160174279	4.22	1478804
9	2120.09	314349742	7.88	2338922
10	> 3600		11.73	3528767

Table 12: Performing deadlock analysis on SwimmingPool model from Figure 8. The size is a multiplier of the token count in initial marking and the weight on all inhibitor arcs. Time is measured in seconds.

8.5 Siphon-trap Analysis

We perform experiments on deadlock queries (category **DL**) to examine the performance of siphon-trap analysis. We are interested in comparing our implementation to the siphon-trap analysis of LoLA, and study the contribution of siphon-trap analysis to the currently best configuration of TAPAAL. We expect that siphon-trap analysis will detect some deadlock freedom in models that are not feasible to verify with our current state space exploration techniques. We also expect faster verification when performing structural reduction prior to siphon-trap analysis. We compare tools configurations **Siphon** and **LSiphon** that both perform siphon-trap analysis (using integer linear program and SAT formulae respectively) without state space exploration. We also compare **StubStruct** with **Best** that performs siphon-trap analysis before switching to state space search using structural reduction, stubborn reduction (it also performs formula simplification, but it has no effect on deadlock formulae).

Table 13 shows how the siphon depth affects the computation time of siphon-trap analysis. As discussed in Section 5, the larger the depth, the more infeasible formulae will be identified, and the more conclusive deadlock freedom answers the tool provides at the cost of more computation. For each timeout we display two metrics: (top) the number of infeasible formulae and (bottom) the number of feasible formulae. In all configurations, a formula is decided feasible if a solution is found, infeasible if no solution is found and the Petri net is deadlock free. In all timeouts we see the largest increase in conclusive answers from the depth of 4 to 8. From 8 to $|P|$ there is almost no increase in conclusive answers. The timeout has not much impact in most configurations, implying that the generated integer linear programs are either very easy or very difficult to solve, with very little in between. This could also suggest that the implementation scales poorly with the size of the Petri net. We highlight the cell that shows what we believe to be a good configuration of depth and timeout for siphon-trap analysis, as it provides 24 conclusive exclusive answers in 5 seconds using a depth of 8. This is the setting that will be used in the remaining experiments. We include a column for configuration **Siphon** to show the difference of doing only siphon-trap analysis to performing structural reduction before siphon-trap analysis (**StructSiphon**). We see it has a positive effect on the number of conclusive answers, but only on instances with a timeout of 10 or less. Table 13 also provides perspective on the performance of our implemented siphon-trap formula compared to LoLA. The siphon-trap implementation of LoLA also only works on 1-weighted Petri net. We see that out of these models, the **LoLA** solves satisfiablity of considerably more SAT formulae than **StructSiphon** does of LP formulae. The SAT encoding build by LoLA may be more efficient than the generated integer linear program, or the MiniSAT solver is simply faster than lp_solve for this kind of problem.

We compare **Best** to **StubStruct** in Table 14 using our scoring metrics. The siphon-trap analysis provides just 2 new answers to the collective set of techniques developed. It is possible that there are more instances that will be exclusively answered by the siphon-trap analysis if more timeout was given, however, this will imply more overhead on all the instances where the siphon-trap property cannot be decided within reasonable time. There are 5 instances where siphon-trap analysis provides a faster answer.

TO	**configuration / depth**						
	StructSiphon					**Siphon**	**LSiphon**
	4	8	12	16	$\lvert P\rvert$	$\lvert P\rvert$	$\lvert P\rvert$
1	13	20	20	20	20	13	33
	120	78	69	62	35	33	64
3	16	20	20	20	20	16	33
	126	89	76	72	44	36	75
5	16	24	20	20	21	16	33
	129	94	81	79	46	38	79
10	16	24	20	21	21	16	35
	132	98	89	84	51	42	80
30	17	25	26	25	25	25	36
	138	112	103	95	56	47	89
60	17	25	27	27	28	28	37
	144	115	106	104	59	47	95

Table 13: Siphon-trap analysis on the 224 1-weighted models, showing performance when combining a range of timeout (TO, measured in seconds) and siphon depth settings. The timeouts for **StructSiphon** and **Siphon** are measured on siphon-trap analysis alone. The **LSiphon** configuration cannot be compared directly, as we do not have access to the concrete timing information of the siphon-trap analysis and thus the timeout is measured in total execution time. For each timeout we show two rows: the number of infeasible formulae and the number of feasible formulae.

Best vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
DL	2	0	12	7	20	1	9	115

Table 14: Pairwise score comparison on deadlock queries, showing the performance difference of adding siphon-trap analysis to the stubborn and structural reduction techniques. There are 313 total queries, 224 of which are 1-weighted.

8.6 Formula simplification

We perform experiments on all cardinality and fireability categories (**CC**, **RC**, **CF**, **RF**) to examine the performance of formula simplification. We are interested in comparing the implementation to the state equations technique of TAPAAL 2.0. We expect that formula simplification will provide more conclusive answers than the state equations of TAPAAL 2.0. We also expect that stubborn and structural reduction will benefit in performance when simplifying the formula prior to the reductions.

Table 15 gives an overview of the number of solved queries by all the configurations that we will refer to in this section.

Table 16 show statistics of formula simplification performance. We see a considerable number of trivially solved queries in the cardinality categories. Furthermore, we see an average reduction effectiveness of $\sim$50% queries, which means we half the number of nodes in the queries on average. In fireability we do not see a significant reduction performance.

Table 17 show a pairwise score comparison of **SimpOnly** and **SE**. We see that in cardinality, the simplification algorithm covers all the answers of the previous state equations technique. However, in fireability, the state equations are much superior, which indicates that there is still room for improvement in the simplification technique. Many of the missing trivial formulae have later shown to be because of disjunction. If two subformulae are trivial in disjunction, we do not discover it.

Table 18 show pairwise score comparisons of **Base**, **Simp**, **Stub**, **Struct**, **SimpStub**, **SimpStruct**, **StubStruct**, and **Best**. We see in all comparisons that formula simplification generally provides more exclusive answers, uses less time, explores less states, and consumes more memory. When comparing **StubStruct** with **Best** in Table 18b the benefit of adding formula simplification to the previously best performing configuration becomes apparent. The simplification procedure consumes notably more memory.

We are interested in how stubborn and structural reduction is affected by the formula simplification. To reason about this, we have excluded answers that are simplified to trivial such that we are left with formulae that are non-trivial and possibly reduced in size. We show these comparisons with these formulae in Tables 18e and 18f, where we compare stubborn and structural reduction to their simplified counterpart. On reachability, stubborn and structural reduction both improve in exclusive answers when simplifying the queries prior to state space exploration. However, on fireability, the performance actually worsens. This may be due to the same issues presented earlier when simplifying fireability formulae.

In Table 19 we highlight how the configurations perform on the most difficult instances. We have chosen the 5 hardest instances from **RC** and **RF** where all configuration provides an answer.

	Number of queries answered									
cat.	**queries**	**Base**	**SE**	**Simp**	**Stub**	**Struct**	**SimpStub**	**SimpStruct**	**StubStruct**	**Best**
RC	1565	817	908	1310	961	879	1376	1334	996	1396
RF	1565	1082	983	1129	1184	1125	1224	1170	1212	1251
subtotal	3130	1899	1891	2439	2145	2004	2600	2504	2208	2647
CC	1565	964		1211						
CF	1565	934		952						
total	6260	3797	1891	4602	2145	2004	2600	2504	2208	2647

Table 15: Number of queries solved by each configuration. We only count the queries where the each configuration can utilise all its techniques, e.g. we do not apply stubborn reduction to CTL formulae.

				med. query size		**avg. query size**	
category	**trivial**	**med. red. %**	**avg. red. %**	**before**	**after**	**before**	**after**
CC	438	25	41.5	15	6	899	477
CF	66	0	9.7	30	25	468	385
RC	675	50	50.6	16	4	750	308
RF	166	0	16.8	36	24	242	109

Table 16: Performance of formula simplification. Trivial counts the number of queries simplified to either *true* or *false*. Med. red. % and avg. red. % counts the median (and average) simplification percentage in formula size of all queries. Query size counts the number of nodes in the query tree before and after applying simplification.

SE vs SimpOnly								
cat.	**exclusive**		**time**		**memory**		**trivial**	
RC	0	484	96	148	0	126	191	675
RF	100	15	325	186	43	50	251	166
total	100	499	421	334	43	176	442	841

Table 17: Pairwise score comparison of **SimpOnly** and **SE**. In each category **RC** and **RF** there are 1565 queries. Trivial measures the number of queries were simplified to trivial by using formula simplification (left) and state equations (right).

Base vs Simp								
cat.	**exclusive**		**time**		**states**		**memory**	
CC	2	250	81	278	1	290	144	199
CF	11	31	253	50	0	54	260	44
RC	5	497	63	535	21	224	267	125
RF	48	99	323	135	96	173	518	55
total	66	877	720	998	118	741	1189	423

(a)

StubStruct vs Best								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	4	404	83	453	27	346	339	152
RF	35	74	330	101	92	175	615	45
total	39	478	413	554	119	521	954	197

(b)

Stub vs SimpStub								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	3	418	71	468	25	319	335	119
RF	40	80	320	104	87	165	606	37
total	43	498	391	572	112	484	941	156

(c)

Struct vs SimpStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	5	460	74	517	25	277	288	143
RF	46	91	333	120	99	181	539	48
total	51	551	407	637	124	458	827	191

(d)

Stub vs SimpStub, excluding trivials								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	3	15	65	29	25	54	335	12
RF	40	11	317	30	87	76	603	9
total	43	26	382	59	112	130	938	21

(e) These scores consider only queries that were not simplified to trivial by **SimpStub**.

Struct vs SimpStruct, excluding trivials								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	5	14	73	28	25	55	288	17
RF	46	14	330	38	99	100	536	19
total	51	28	403	66	124	155	824	36

(f) These scores consider only queries that were not simplified to trivial by **SimpStruct**.

Table 18: Pairwise score comparisons. In each category **CC**, **CF**, **RC** and **RF** there are 1565 queries.

	Stub			**Struct**			**Simp**			**SimpStub**			**SimpStruct**			**StubStruct**			**Best**		
model	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**	**time**	**mem**	**states**
M1	197	609	19689	887	1612	59408	0	3.38	0.00	0	3.38	0.00	0	3.38	0.00	56	183	6145	0	3.38	0.00
M2	235	1743	35337	825	4608	124235	853	4608	124235	235	1745	35337	854	4608	124235	233	1743	35337	232	1743	35337
M3	0	4.14	32.8	0	3.19	0.58	772	6609	212825	0	4.93	32.8	0	3.99	0.58	0	3.63	10.4	0	4.24	10.4
M4	724	1426	280	746	1426	280	254	2001	60.0	251	2001	60	286	2001	60.0	770	1426	280	289	2001	60
M5	5	14.7	257	0	5.53	57.8	758	1407	43564	5	15.3	257	0	5.86	53.8	0	4.50	8.21	0	5.09	8.21
M6	690	151	37	702	149	36.6	894	150	36.6	885	149	36.6	881	149	36.6	700	149	36.6	894	148	36.6
M7	604	414	40	655	414	40.0	826	414	40.0	832	414	40.0	859	414	40.0	655	415	40.0	866	415	40.0
M8	759	551	3.24	800	552	3.43	5	552	0.00	5	552	0.00	5	552	0.00	829	551	3.43	5	551	0
M9	0	17.6	0.24	0	16.7	0.24	728	9042	0.24	713	9042	0.24	762	9042	0.24	0	17.5	0.24	703	9042	0.24
M10	587	733	27750	271	744	27951	327	773	27951	649	756	27750	324	772	27951	592	734	27750	681	756	27750

Table 19: The most difficult instances for at least one of the included configurations, where all configurations provide answer. Time is in seconds, memory is in MB, states is in number of thousands. Models (name, category, query index): **M1** ResAllocation-PT-R003C015, **RC**, 1 **M2** PermAdmissibility-PT-05, **RC**, 4 **M3** HouseConstruction-PT-200, **RC**, 3 **M4** Philosophers-PT-010000, **RC**, 1 **M5** CloudDeployment-PT-2b, **RC**, 4 **M6** DatabaseWithMutex-PT-40, **RF**, 2 **M7** Philosophers-PT-010000, **RF**, 4 **M8** GlobalResAllocation-PT-05, **RF**, 2 **M9** BridgeAndVehicles-PT-V80P50N10, **RF**, 3 **M10** DNAwalker-PT-08ringLL, **RF**, 2

8.7 Best vs. Best

We perform experiments on all queries from all categories. There are 5008 queries from each category **CC**, **CF**, **RC**, and **RF**, and 313 queries from category **DL** to compare the overall performance of TAPAAL and LoLA. We are interested in comparing the scoring metrics between the both tools using all described techniques, i.e. configurations **Best** and **LoLA**.

Table 20 show the number of queries solved by both configurations. We see that **LoLA** solves 264 more queries on all queries from the reachability category, whereas **Best** solves 1156 more queries from the CTL category. In total, **Best** 892 more queries.

Pairwise score comparison between the two configurations is shown in Table 21. The largest difference in exclusive answers is in category **CC**, where **Best** exclusively solves more than a fifth of all queries. In every other categories, **LoLA** perform faster on more queries, specifically close to one third of all queries on category **RF**. We see that the number of states explored differ widely on both configurations, however, **LoLA** generally explores fewer states on all categories.

In Table 22 we highlight how the two configurations perform on the most difficult instances. We have chosen the two hardest instances from each category where both configurations provides an answer. Instance **M3** is a case that show that the formula simplification of **LoLA** does not simplify the same formulae to trivial as **Best** does. The reverse scenario is also present in the raw data. On **M7** we see a case where a full state space is required to verify the query, and that **Best** and **LoLA** have explored the exact same number of states. In this case, **Best** is much more memory consuming than **LoLA**, but verifies the query in less than half the time.

		queries solved	
cat.	**queries**	**Best**	**LoLA**
DL	313	241	255
RC	5008	4476	4507
RF	5008	4011	4230
CC	5008	3976	3097
CF	5008	3181	2904
total	20345	15885	14993

Table 20: Number of solved queries. In each category **RC**, **RF**, **CC** and **CF** there are 5008 queries. In category **DL** there are 313 queries.

Best vs LoLA									
cat.	**queries**	**exclusive**		**time**		**states**		**memory**	
DL	313	3	17	18	28	82	155	106	111
RC	5008	281	312	557	691	2175	2986	3105	772
RF	5008	250	469	336	1484	829	2778	2353	1133
CC	5008	1202	323	1421	633	1047	1934	1051	1522
CF	5008	698	421	852	1170	385	1900	443	1879
total	20345	2434	1542	3184	4006	4518	9753	7058	5417

Table 21: Pairwise score comparisons. In each category **RC**, **RF**, **CC** and **CF** there are 5008 queries. In category **DL** there are 313 queries.

	Best			**LoLA**		
model	**time**	**mem**	**states**	**time**	**mem**	**states**
M1	1093	927	25402	133	200	1718
M2	26	63	403	1037	4303	2105
M3	0	2.99	0	882	13279	168208
M4	9	518	0	874	1568	0
M5	897	95538	3.21	0	6.11	4.34
M6	880	148	36.6	446	3103	0.01
M7	344	10826	32961	894	8826	32961
M8	0	6.61	0.26	886	7797	72061
M9	880	8284	22020	665	39.2	21925
M10	876	9072	5569	230	2315	5303

Table 22: The most difficult instances for at least one of the included configurations, where both configurations provide answer. Time is in seconds, memory is in MB, states is in number of thousands. Models (name, category, query index): **M1**: CloudDeployment-PT-5a, **DL**, 1. **M2**: PolyORBLF-PT-S04J06T10, **DL**, 1. **M3**: GPPP-PT-C1000N0000000010, **RC**, 14. **M4**: TokenRing-PT-040, **RC**, 14. **M5**: SafeBus-PT-03, **RF**, 9. **M6**: DatabaseWithMutex-PT-40, **RF**, 2. **M7**: BridgeAndVehicles-PT-V50P50N20, **CC**, 16. **M8**: DNAwalker-PT-13ringRLLarge, **CC**, 12. **M9**: SimpleLoadBal-PT-20, **CF**, 3. **M10**: BridgeAndVehicles-PT-V80P50N10, **CF**, 15.

9 Conclusion

The feasibility of model checking is strongly dependent on the efficiency of the verification technique. Our experiments show that many real-world systems generates an unmanageable amount of states, and that certain countermeasures have to be taken, in order to provide meaningful information. We have also seen, that many formulae contains trivial subformulae, that can be answered in the initial state, possibly because they were automatically generated.

Our work in this Master's thesis and in our pre-specialisation project, has indeed confirmed that there exists many effective methods and approaches to model checking. The techniques used in this thesis have improved verification performance of the state of the art model checker TAPAAL. Some techniques are specifically good in verifying certain properties and models, and some techniques even improve when used in combination.

We have extended our stubborn reduction implementation, to handle Petri nets with inhibitor arcs, and we have restructured and improved the code that generates the set of interesting transitions. Experiments show that our stubborn reduction implementation is competitive on the set of models from MCC'17. The inhibitor extension to stubborn reduction was showcased on a scaled set of model instances, and demonstrate great performance. The stubborn and structural reduction techniques generally both benefit from being applied in combination, but experiments show that the techniques conflict on some models, which can lead to computational overhead.

We presented an interpretation for deciding the siphon-trap property, expressed as an integer linear program. Experiments showed that structural reduction prior to siphon-trap analysis improved the number of decided instances of the siphon trap property, as the number of places and transitions is reduced. Lastly, we saw that the satisfiability formula that LoLA uses for siphon-trap analysis, performed better than our integer linear program implementation.

State equations have previously shown to be effective when used in combination with explicit analysis methods. We extended the theory to handle not only reachability formulae, but also CTL formulae. In our experiments, we saw great improvement on cardinality formulae. However, there were a significant number of fireability formulae that were not able to be simplified to trivial formulae, even though they were actually trivial. We believe the simplification procedure can be improved in this aspect. The formula simplification greatly improved the overall performance of TAPAAL, both by providing trivial answers and reducing formulae such that the stubborn reduction improves by generating smaller interesting sets.

We examined the combined performance of every implemented technique in TAPAAL and compared it with LoLA. On all categories combined, TAPAAL provided more exclusive answers and used less memory. LoLA performed faster verification and generally explored fewer states. Specifically, LoLA performed better on reachability formulae and TAPAAL performed better on CTL formulae.

9.1 Future Work

There are several extensions to the Petri net formalism that we can consider as future work. We already covered the extension including inhibitor arcs, but another possibility is coloured Petri net. In a coloured Petri net, each token is equipped with a 'colour', which is of a certain type specified by the given place [12] the token is in. Coloured Petri nets can achieve a more compact representation when compared to basic Petri nets, and are more analogous to programming languages as token colours correspond to data types [12].

How stubborn reduction can be applied to the unique structure of workflow nets is also a possibility. It would be interesting to investigate if the structural properties of workflow nets can be exploited to verify soundness faster using stubborn reduction.

We applied the formula simplication procedure to CTL formulae, but the stubborn reduction procedure only works on reachability formulae, or CTL formulae that have been simplified to reahcability formulae. The next logical step is to investigate how stubborn reduction can be applied to CTL formulae.

As was demonstrated, there exists nondeterminism in both the generation of interesting sets of transitions and the stubborn closure algorithm, which implies that there, for a given state, can exist several valid stubborn sets. It is interesting to investigate how to determine which choice is the best, whenever we are given a choice in the procedures.

The experiments showed that the integer linear programming interpretation for deciding the siphon-trap property did not live up to out expectations. It generally performed worse than the SAT implementation presented by LoLA. Further work needs to be done to optimise both the system of equations and its implementation, e.g. by reducing the number of constraints and decision variables. Furthermore, investigating how the siphon-trap property can be extended to include Petri net formalism extensions such as inhibitor and weighted arcs would be the next logical steps, as especially the 1-weighted requirement is limiting.

The formula simplification procedure did not provide the expected improvement on fireability queries, and in fact performed worse when compared to the state equations of TAPAAL 2.0. All fireability formulae generally have the same form because they are converted to cardinality formulae, so it might be the stucture of the converted formula that is unsuited for the current formula simplification procedure.

An improvement to the formula simplification is to examine if two non-trivial subformulae are trivial in disjunction. This can be done by negating the disjunction formula and then prove the impossibility of its linear programs. If the negated disjunction is impossible, then the original disjunction must be trivially satisfied. The experiments indicate that there is room for improvement when considering the memory consumption.

10 Bibliographical Remarks

We reuse and extend some of the work in our pre-specialisation thesis [16]. The CTL logic in Section 2.1 is based on the already defined reachability marking properties in [16], and we now describe the definition in the context of LTS. We reuse the definition, semantics, graphical notation and example of Petri net, and extend them with inhibitor arcs. Section 3 is based on Section 3 from [16], and includes minor modifications and new examples. Section 4 is based on Section 4 from [16] and extends definitions and proofs to consider inhibitor arcs when generating the set of interesting transitions, and in the reachability preserving closure, as shown in Table 1, and Proposition 1. The proofs are reused and updated accordingly.

References

1. LoLA Formula preprocessing. Section 3.3.3.3. `https://www2.informatik.hu-berlin.de/top/lola/loladoku`. Accessed: 17/2-2017.
2. Model Checking Contest (MCC) 2016. `mcc.lip6.fr`. Accessed: 6/12-2016.
3. Michel Berkelaar, Kjell Eikland, Peter Notebaert, et al. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.
4. Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.
5. Edmund M. Clarke, E Allen Emerson, and A Prasad 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.
6. Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H Møller, and Jiří 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*, pages 492–497. Springer, 2012.
7. Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5:8th, 2005.
8. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Form. Meth. in Syst. Design*, 16:159–189, 2000.
9. Michel Henri Théodore Hack. Analysis of production schemata by petri nets. Technical report, DTIC Document, 1972.
10. Monika Heiner, Christian Rohr, and Martin Schwarick. Marcie–model checking and reachability analysis done efficiently. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 389–399. Springer, 2013.
11. Jonas F. Jensen, Thomas Nielsen, Lars K. Oestergaard, and Jirı Srba. TAPAAL and Reachability Analysis of P/T Nets. In *Transactions on Petri Nets and Other Models of Concurrency XI*, pages 307–318. Springer, 2016.
12. Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.
13. Peter Gjøl Jensen, Kim Guldstrand Larsen, Jiří Srba, Mathias Grund Sørensen, and Jakob Haar Taankvist. *Memory Efficient Data Structures for Explicit Verification of Timed Systems.* NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 – May 1, 2014. Proceedings, pages 307–312, Springer International Publishing, 2014.
14. Marcin Kalicinski. Rapidxml, 2009.
15. Lars Michael Kristensen, Karsten Schmidt, and Antti Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design*, 29(3):215–251, 2006.
16. Jakob Dyhr Mads Johannsen and Frederik Meyer Bønneland. Reachability analysis: A stubborn approach. 2016.
17. George L Nemhauser and Laurence 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.
18. Olivia Oanea, Harro Wimmel, and Karsten Wolf. New algorithms for deciding the siphon-trap property. In *International Conference on Applications and Theory of Petri Nets*, pages 267–286. Springer, 2010.

19. Carl Adam Petri. Kommunikation mit automaten. 1962.
20. Karsten Schmidt. Stubborn sets for standard properties. In *International Conference on Application and Theory of Petri nets*, pages 46–65. Springer, 1999.
21. Karsten Schmidt. Integrating low level symmetries into reachability analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 315–330. Springer, 2000.
22. Karsten Strehl and Lothar Thiele. *Symbolic model checking using interval diagram techniques*. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich (ETH), 1998.
23. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
24. Antti Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer, 1989.
25. Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.
26. Antti Valmari and Henri Hansen. Stubborn set intuition explained. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE*, pages 213–232, 2016.
27. Harro Wimmel and Karsten Wolf. Applying CEGAR to the Petri net state equation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 224–238. Springer, 2011.
28. Karsten Wolf. Running LoLA 2.0 in a Model Checking Competition. In *Transactions on Petri Nets and Other Models of Concurrency XI*, pages 274–285. Springer, 2016.