PeerJ Computer Science

starMC: an automata based CTL* model checker

Elvio Gilberto Amparore, Susanna Donatelli and Francesco Gallà

Dipartimento di Informatica, Università degli Studi di Torino, Torino, Italy

ABSTRACT

Model-checking of temporal logic formulae is a widely used technique for the verification of systems. CTL* is a temporal logic that allows to consider an intermix of both branching behaviours (like in CTL) and linear behaviours (LTL), overcoming the limitations of LTL (that cannot express "possibility") and CTL (cannot fully express fairness). Nevertheless CTL* model-checkers are uncommon. This paper presents (1) the algorithms for a *fully symbolic automata-based* approach for CTL*, and (2) their implementation in the open-source tool starMC, a CTL* model checker for systems specified as Petri nets. Testing has been conducted on thousands of formulas over almost a hundred models. The experiments show that the fully symbolic automata-based approach of starMC can compute the set of states that satisfy a CTL* formula for very large models (non trivial formulas for state spaces larger than 10^{480} states are evaluated in less than a minute).

Subjects Theory and Formal Methods, Software Engineering
Keywords Model-checking, System verification, Petri nets, CTL* logic, Tools, Buchi automata

INTRODUCTION

Temporal logics like LTL (*Pnueli, 1977*), for linear behaviour, and CTL (*Clarke & Emerson, 1981*), for branching behaviour, have been successfully used to specify sequential and concurrent systems, and have been widely adopted in many industrial contexts: see, *e.g.* (*Eisner & Fisman, 2016*) or the success stories of the model checkers SPIN (*Holzmann, 2004*) and nuSMV (*Cavada et al., 2014*). Although LTL and CTL model-checking procedures are known to be quite expensive, since they incur in the so-called "state-space explosion" problem, the use of techniques based on decision diagrams allows, in many cases, to solve industrial-size systems (*Burch et al., 1992*). In this paper we use "symbolic" to refer to any technique based on some form of decision diagram.

CTL* (*Emerson & Halpern, 1983*, *1986*) is a temporal logic that allows to express linear and branching behaviours in a single property. CTL* allows to go beyond the linear/ branching dichotomy (or LTL/CTL), and it is considered a powerful specification language for discrete events dynamic systems. CTL* is known to be strictly more expressive than CTL and LTL. Rozier, in her 2011 survey (*Rozier, 2011*). Considers CTL* as an adequate logic to overcome LTL and CTL limitations, and in that survey she observes that "*the lack of industrial model-checking tools that accept CTL* specifications is a deterrent to the use of this logic*" (*Rozier, 2011*, p.175, end of Sec. 3.2.3), and indeed the situation has not changed much since then.

Submitted 27 August 2021
Accepted 29 November 2021
Published 24 February 2022

Corresponding author
Susanna Donatelli,
susanna.donatelli@unito.it

Academic editor
Mohammad Reza Mousavi

Additional Information and Declarations can be found on page 35

DOI **10.7717/peerj-cs.823**

CTL* properties are of practical interest, and CTL* model checking has been extensively studied in the past. However, as pointed out in *Amparore, Donatelli & Gallà (2020b)*, very few CTL* model checkers exist today, despite its usefulness in specifying both recurrent behaviours and possibility in the same property (something happening infinitely often and for all behaviour, a certain event/state is reachable), or to explicitly express fairness constraints. LTL can express both fairness constraints and recurrent behaviours, but this is not possible in CTL. Instead, CTL can express possibility, which is not possible in LTL.

There are different ways of verifying CTL* properties: with an ad-hoc model-checker based on the identification and verification of LTL sub-formulae (*Emerson & Lei, 1987*); or through a translation from CTL* to μ-calculus (*Kozen, 1983*), as defined in *Dam (1990*, *1994)* and revised in *Cranen, Groote & Reniers (2011)*. μ-calculus is indeed known to subsume CTL*, but translated formula may be unintuitive to understand, and the translation could lead to an exponential growth of the expression terms.

The Petri net (or PN for short) is a formalism well-suited for the description and the analysis of Discrete Events Dynamic System (DEDS). They have been successfully used in a large variety of application fields, including industrial production (*Zurawski & Zhou, 1995*; *van der Aalst, 1994*), business workflow (*Aalst, 1998*), digital circuits synthesis and analysis (*Kondratyev et al., 1998*) and system biology (*Koch, 2019*). This success is mainly due to the particular mixture of ease of specification and good support for the analysis. Analysis techniques can prove properties using only the structure of the PN, or its state space (which is typically exponential in the size of the structure and of its initial state).

The model checking of temporal properties for Petri nets is supported by numerous tools (*Kordon et al., 2019*). In the last decade the model checking of PN has seen a boost in interest, possibly motivated by the lively Model Checking Competition (MCC) (*Kordon et al., 2019*) and advances in DD-based implementation of state space exploration based on saturation (*Ciardo, Lüttgen & Siminiceanu, 2001*).

The contribution of this paper is to clearly identify the algorithms and the steps of a fully symbolic procedure for computing the *sat-set* of a CTL* formula (the set of states of a given model that satisfy the formula). The implementation of such a procedure for systems specified as Petri nets results in `starMC`, a tool that computes the sat-set of a CTL* formula using a Büchi-based model checking algorithm. Given a CTL* formula, the algorithm identifies the LTL sub-formulae of maximal length and uses the Spot library (*Duret-Lutz, 2014*; *Duret-Lutz et al., 2016*) to translate each of them into a Büchi automaton. The sat-set of each LTL sub-formula is then built from the sat-set of the $E_{\text{fair}}G(\mathit{true})$ on the synchronized product of the model state space and the generated automaton. States, transitions among states, and the synchronized products are all encoded as Decision Diagrams, using the `Meddly` (*Babar & Miner, 2010*) library. `starMC` can be run as a stand-alone tool or as part of the `GreatSPN` (*Amparore et al., 2016*) tool suite through the built-in graphical interface (*Amparore, 2014*).

We could only find another Petri net tool that can deal directly with CTL*: `LTSmin` (*Kant et al., 2015*). This tool translates CTL* into μ-calculus, using the procedures defined in *Dam (1990)*. The search for a CTL* model-checker was not more successful when considering input languages other than Petri Nets.

Validation of `starMC` has been achieved taking advantage of the MCC benchmark (*Kordon et al., 2019*) that comprises models, LTL and CTL formulae, and associated truth values for the models' initial state.

The benchmark has been enriched with CTL* formulae, derived from the available CTL ones by randomly omitting path quantifiers. Sat-sets of the LTL, CTL, and CTL* formulae have been compared against the sat-sets obtained from the CTL* module of `LTSmin`. Sat-sets of CTL formulae have also been checked against the CTL model checker in the distribution of `GreatSPN`.

A motivation for building `starMC` was teaching and training: to provide students with a uniform environment in which to reason on, and to experiment with, LTL, CTL and CTL* specifications. This motivation follows an earlier re-shaping of `GreatSPN` to support teaching (*Amparore & Donatelli, 2018b*). But the main motivation was to investigate a number of open research questions: **R1**: it is possible to realize an efficient and fully symbolic implementation of CTL* based on Büchi automata that can check models of industrial interest? **R2**: the approach of R1 is any better than a translation to μ-calculus? **R3**: variable ordering techniques that exploits the PN structure, like the ones developed in *Amparore et al. (2019)* for state-space exploration, can be successfully applied to CTL* model-checking? **R4**: can a Büchi-based approach favour the formulation of counterexamples and witnesses? This paper addresses R1 and R2, and uses the techniques presented in *Amparore et al. (2019)* for variable ordering. An extensive comparison of different variable order heuristics, as well as the counter-examples generation of R4, are left for future work.

The `starMC` tool basic principles and structure were first presented in a (demo) paper (*Amparore, Donatelli & Gallà, 2020b*): here we take a deeper, and at the same time broader, view on CTL* model-checking and on its implementation in `starMC`. We provide distinct logical and implementation views of the model-checker, thus separating the algorithms from their fully symbolic implementation. The two views and the chosen level of detail, together with a fully open-source code, are meant to ease the scientific and implementation work needed to extend other verification engines with a CTL* model-checker, or to add to `starMC` additional techniques, for example for optimization or parallelization. We have also added a deeper investigation on previous work to precisely link the approach used with their original sources of inspiration. This paper also introduces a user view-point on `starMC`, for those interested only in using the tool. Finally, the tool validation has been improved, including less stringent execution time limits, more models and properties, and a deeper investigation of the obtained results.

The paper is organized as follows: "Background" introduces basic definitions and background, "starMC: the outer view" provides an overview of `starMC` from an user view-point, "starMC: the logical view" describes, at a logical level, the CTL* model-checking procedure, followed by, in "Previous Work", a discussion and comparison with previous work. The DD-based implementation of the algorithms of "starMC: the logical

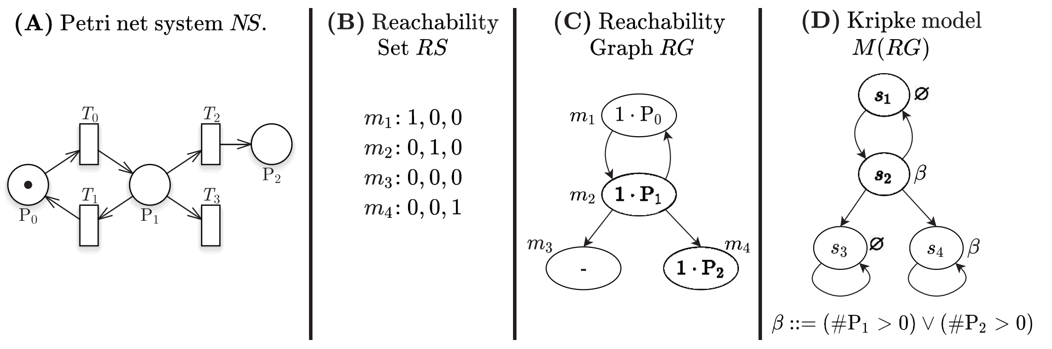


Figure 1 A Petri net, its RS and RG, and the corresponding Kripke model.
Full-size DOI: 10.7717/peerj-cs.823/fig-1

view" is discussed in "starMC: The Inner View". "Testing Results" describes the tests that have been conducted, and "Conclusion" concludes the paper.

BACKGROUND

This section briefly recalls the PN formalism and the CTL* logic. It also introduces the main feature of the `GreatSPN` tool and of the Meddly and Spot libraries.

Petri nets

A *place-transition (P/T)* Petri net N is defined (Murata, 1989) as a tuple $N = \langle P, T, A, W \rangle$, where P is the set of places, T is the set of transitions, $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, $W : A \longrightarrow \mathbb{N}_{\geq 1}$ is the arc weight function. A Petri net system is the pair $NS = \langle N, m_{\text{init}} \rangle$ where $m_{\text{init}} : P \longrightarrow \mathbb{N}$ is the initial marking. Markings represent states of the system, *i.e.* assignments of tokens to places. A transition $t \in T$ is *enabled* if and only if all input places p of t contain at least $W(p, t)$ tokens. The *firing* of t removes $W(p, t)$ from all input places p and adds $W(t, p')$ to each output place p'. Notation $m \xrightarrow{t} m'$ indicates the firing of t in marking m, which leads to marking m'. A firing sequence σ for marking m is a sequence of transitions $t_0, t_1, \ldots$ for which it exists a sequence of markings $m_0, m_1, \ldots$ such that $m = m_0$ and, $\forall i \geq 0 : m_i \xrightarrow{t} m_{i+1}$. Accordingly, the sequence of markings $\pi = m_0, m_1, \ldots$ identifies *a path* of NS starting in m_0, with $\pi[i] = m_i$. We indicate with $\pi[i{\cdot\cdot}]$ the suffix of π starting in m_i. We define $\mathscr{P}^{\infty}_{M}(m)$ as the set of all infinite paths starting in m. From now on we shall interchangeably use the terms "marking" and "state". The *reachability set* (RS) of a Petri net system NS is the set of all markings reachable from m_{init}, $RS(NS) = \{m \mid \exists_{\sigma \in T*}\ m_{\text{init}} \xrightarrow{\sigma} m\}$. A *reachability graph* (RG) of a PN is a tuple $RG(NS) = \langle RS, \mathscr{E}, m_{\text{init}} \rangle$ where: RS is the set of all markings reachable from m_{init}, $\mathscr{E} = \{(m, t, m') \in RS \times T \times RS \mid m, m' \in S, t \in T, m \xrightarrow{t} m'\}$ is the set of state transitions. If there is no risk of confusion, the NS indication is usually dropped and we write RS and RG.

Figure 1A shows a Petri net with 3 places and 4 transitions, the RS built from the initial state $m_{\text{init}} = m_1$, in Fig. 1B, and the corresponding RG, in Fig. 1C.

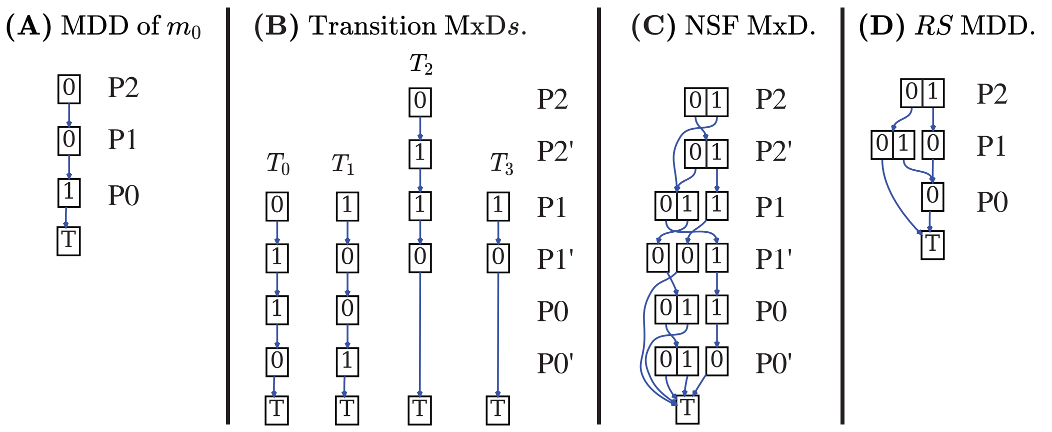


Figure 2 **Decision diagrams for the state space generation of the net in Fig. 1A.**
Full-size DOI: 10.7717/peerj-cs.823/fig-2

A *colored* Petri net is a P/T net in which tokens have identities, to allow for a parametric and more compact description of systems. Each colored Petri net can be *unfolded* into a P/T net of isomorphic RG.

Decision diagrams for state space representation

Decision Diagrams (DD) are a well-known data structure to efficiently encode functions as well as large sets of structured data through their characteristic function. A Binary DD (BDD) encodes a boolean function and a Multivalued DD (MDD) encodes an integer function. A MDD can be used to encode a RS by associating each place of the net to a level in the MDD and ensuring that each path in the MDD corresponds to a reachable state (and *vice-versa*). Given the example Petri net of Fig. 1A, the MDD of the initial state m_1 is depicted in Fig. 2A, and the one of RS in Fig. 2D. DD levels encode the token counts of each Petri net place, and appear in the order $P2$, $P1$, $P0$ (top-down). So the rightmost path in the MDD of RS corresponds to the reachable state $[m(P2) = 1, m(P1) = 0, m(P0) = 0]$, or [1, 0, 0] for short. The depicted MDDs are fully-reduced (when all edges out of a node lead to the same down node, the node is removed and by-passed), so the leftmost path of this MDD encodes both [0, 0, 0] (no token in any place) and [0, 0, 1] (one token in $P0$).

A DD encoding a relation function is called *matrix diagrams* (M×D). These kind of DDs are used in `starMC` to represent transitions among states due to the firing of Petri net transitions. An M×D has twice the levels of an MDD, and each pair of levels (also called the unprimed and primed levels for a variable) encodes the before/after relations of a variable (*i.e.* a place of a Petri net or a location of an automaton). Each of the four M×Ds in Fig. 2B encodes the transformation performed by a single transition of the example net while the M×D in Fig. 2D is the *Next-State-Function* (*NSF*), the union of all the single transition M×Ds. The depicted M×D are identity-reduced, *i.e.* skipping a level pair means that the encoded before-after relation is the identity. The image of this M×D applied to an MDD encoding a state m returns an MDD with all states that can be reached from m by the firing of a single transition. Its fixed point application results in the MDD of the RS. In practice the more efficient *saturation* technique (*Ciardo, Lüttgen & Siminiceanu,* 2001), also based on the NSF, is usually preferred. Note that usually the M×D of the RG is never built and stored, as any info in the RG can be retrieved through the DDs of NSF and RS.

In Fig. 2 places are allocated to MDD (and M×D) levels in alphabetical order. It is well known that the DD *variable order* (the order in which variables are assigned to levels) may greatly influence the size of the resulting DD. The interested reader may find in *Amparore, Donatelli & Ciardo (2020a)* a recent study on various variable order heuristics that have been successfully applied to encode state spaces of Petri nets.

The temporal logic CTL*

CTL* (*Emerson & Halpern, 1983*, *1986*) is a temporal logic in which path operators can be quantified or not, and it therefore allows to freely mix linear and branching reasoning. It subsumes the linear logic LTL (*Pnueli, 1977*) and the branching logic CTL (*Clarke & Emerson, 1981*), that are known to have a different expressive power, although with non-empty intersection.

Definition 1 (CTL* syntax). *CTL* formulae are the formulae* Ψ *inductively defined by:*

$$\Psi ::= a \mid \Psi \wedge \Psi \mid \neg\Psi \mid E\phi \mid A\phi$$
$$\phi ::= \Psi \mid \phi \wedge \phi \mid \neg\phi \mid X\phi \mid \phi\, U \phi$$

where $a \in AP$ *and* AP *is a set of atomic propositions. The rules* Ψ *and* ϕ *define* state *and* path *formulae, respectively.*

Note that in principle just one type of quantification is needed, as $A\phi = \neg E \neg\, \phi$. The BNF grammar of CTL* is ambiguous, as boolean expressions appear both as state and path formulae. For instance, the expression $a_1 \wedge a_2$, with $a_1, a_2 \in AP$, is both a state formula (rule $\Psi \wedge \Psi$) and a path formula (rule $\phi \wedge \phi$ where $\phi \rightsquigarrow \Psi$ and $\Psi \rightsquigarrow a$).

The satisfaction relation for a CTL* formula is given over a Kripke model $M = (S, E, L)$ with an associated set AP of atomic propositions, where S is a finite and non-empty set of states, $E : S \longrightarrow 2^S$ is the total successor function, $L : S \longrightarrow 2^{AP}$ is a labelling function, and $L(s)$ is the set of atomic propositions that holds in s. The set of paths starting in s, $\mathscr{P}^{\infty}_{M}(s)$ is defined as for Petri nets. It is straightforward to map a reachability graph into a Kripke model.

Definition 2 (Kripke model of an RG). *The Kripke model of a reachability graph* $RG = \langle RS, \mathscr{E}, m_{\text{init}} \rangle$ *is the Kripke model* $M(RG) = \langle RS, \mathscr{E}', L \rangle$, *with* $\mathscr{E}' = \mathscr{E} \cup \{(m, m) : \forall$ *deadlock states* $m \in RS\}$ *being the stuttering of* $\mathscr{E}$, *and* L *a labelling function defined over the markings.*

Stuttering of deadlock states ensures that the paths in the RG are infinite, as required by a Kripke model. Figure 1D shows the Kripke model derived from the RG in Fig. 1C when considering a set of atomic propositions $AP = \{\beta\text{: } (\#P1 > 0 \text{ or } \#P2 > 0)\}$ and the labelling function depicted next to the states. The two states s_3 and s_4 are stuttered, since they correspond to the deadlock markings m_3 and m_4.

Definition 3 (CTL* semantics). *The satisfaction relation of CTL* state formulae is defined by:*

$$\begin{array}{rcc} s \models a & \text{iff} & a \in L(s) \\ s \models \Psi_1 \wedge \Psi_2 & \text{iff} & (s \models \Psi_1) \text{ and } (s \models \Psi_2) \\ s \models \neg\Psi & \text{iff} & \text{not } s \models \Psi \\ s \models A\phi & \text{iff} & \pi \models \phi, \text{ for all } \pi \in \mathscr{P}^{\infty}_{M}(s) \end{array}$$

while the satisfaction relation of path formulae for a path π is:

$$\begin{array}{rcc} \pi \models \Psi & \text{iff} & \pi[0] \models \Psi \\ \pi \models \phi_1 \wedge \phi_2 & \text{iff} & (\pi \models \phi_1) \text{ and } (\pi \models \phi_2) \\ \pi \models \neg\phi & \text{iff} & \text{not } \pi \models \phi \\ \pi \models X\phi & \text{iff} & \pi[1\cdots] \models \phi \\ \pi \models \phi_1 U \phi_2 & \text{iff} & \exists j \geq 0 : (\pi[j\cdots] \models \phi_2 \wedge \\ & & (\forall 0 \leq k < j : \pi[k\cdots] \models \phi_1)) \end{array}$$

A CTL* formula in which all path operators are prefixed by a quantifier is also a CTL formula. To express the relation between CTL* and LTL it is convenient to introduce the following definitions:

Definition 4 (LTL formula). *An LTL formula φ is inductively defined by:*

$$\varphi ::= a \mid \varphi \wedge \varphi \mid \neg\varphi \mid X\varphi \mid \varphi U \varphi$$

It is usually assumed that path formulae over a Kripke model are "implicitly universally quantified". To avoid any confusion we assume that the quantifier is explicitly indicated.

Definition 5 (Quantified LTL formula). *Given an LTL formula φ a quantified LTL formulae Ψ is defined by $\Psi ::= E\varphi \mid A\varphi$*

The GreatSPN tool

`GreatSPN` (*Amparore et al., 2016*; *Amparore & Donatelli, 2018b*) is a collection of Petri nets tools that are integrated in a common framework, with a unified graphical user interface (*Amparore, 2014*). Initially developed for performance evaluation of stochastic Petri nets, the tool was enriched over the years with algorithms for qualitative analysis.

From the graphical interface it is possible to draw a Petri net (colored or P/T), to compose Petri nets, to play the token game to familiarize with the net behaviour, and to verify the model using techniques specific to Petri nets (structural properties like P- and T-semiflows) or to check standard Petri net properties (like absence of deadlocks, boundedness and liveness) as well as CTL properties. The state space exploration algorithm and the CTL model-checker are fully symbolic, built using the MDD library Meddly (*Babar & Miner, 2010*), and a large set of heuristics for variable ordering (*Amparore, Donatelli & Ciardo, 2020a*) that exploit the net structure. Qualitative verification can be integrated by a stochastic verification, as `GreatSPN` includes a model-checker (*Amparore & Donatelli, 2018a*) for the stochastic logic CSL^{TA} (*Donatelli, Haddad & Sproston, 2009*), and the computation of standard performance indices based on efficient analytical solutions of Markov chains and Markov Regenerative Processes, as well as simulation. The tool is open source and it is available from GitHub (https://github.com/greatspn/SOURCES).

The supporting libraries: meddly and spot

Spot 2.0 (*Duret-Lutz et al., 2016*) is a framework which provides, among other features, a wide range of tools to manipulate automata over infinite words and to translate them from LTL propositions. The framework has been maintained for decades and can be considered the state-of-the-art of LTL-to-Büchi translation. Our model checker uses the Spot library to construct a Generalized Büchi Automaton with state-based acceptance (SGBA) from each LTL formula encountered during the verification process. We chose to use the Hanoi-Omega Automata (HOA) format (*Babiak et al., 2015*) to represent such automata. The HOA format is a text-based encoding which supports various derivation of ω-automata. Spot can be downloaded from its home page (https://gitlab.lrde.epita.fr/spotandspot.lrde.epita.fr).

Meddly (*Babar & Miner, 2010*), Multi-terminal and Edge-valued Decision Diagram LibrarY, is an open-source library (https://github.com/asminer/meddly) for decision diagrams, developed at Iowa State University. It supports the construction and manipulation of various kinds of decision diagrams: Binary and Multivalued decision diagrams, matrix diagrams (M×D) and edge-valued DDs, including a built-in function to support efficient state-space construction based on saturation.

STARMC: THE OUTER VIEW

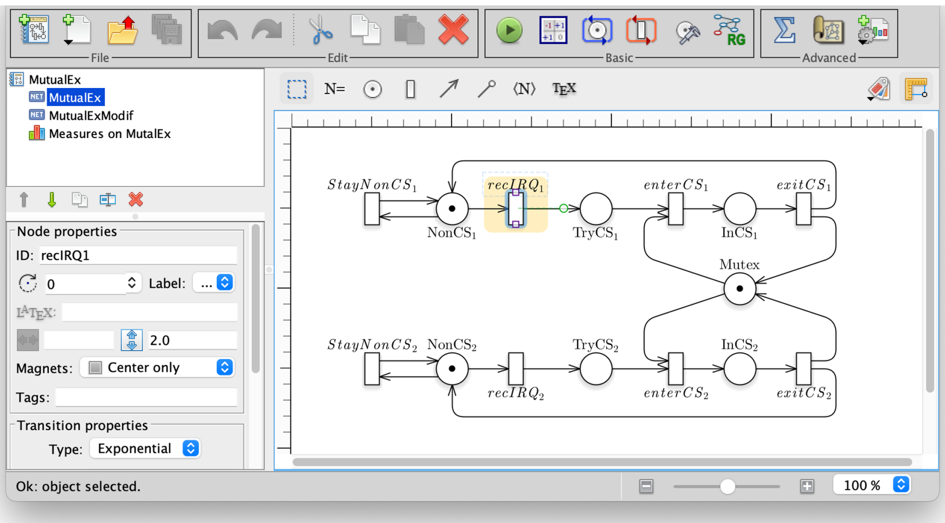
This section describes how to use the tool for checking CTL* properties. We do so through an example, adapted from (*Emerson & Halpern, 1986*) and meant to underline the differences between LTL, CTL, and CTL*, and the advantage of being able to specify and verify CTL* properties. These differences are sometimes subtle and it is indeed convenient to be able to express properties in the three logics in the same tool (and actually from the same window).

The `starMC` tool accepts a CTL* syntax (*Amparore, Donatelli & Gallà, 2020b*) that is more expressive than the minimal one in Definition 1, as it includes more boolean connectors, and the path operators F (in the Future, for some state in the path) and G (globally, for all states in the path). Since CTL* formulae are written in textual mode in the tool, the symbols: &&, ||, !, ,>=, <=, == are used for $\wedge$, $\vee$, $\neg$, $\geq$, $\leq$, $=$, leading to:

$$\Psi ::= AP \mid \Psi \;\&\&\; \Psi \mid \Psi \;||\; \Psi \mid !\Psi \mid E\phi \mid A\phi$$
$$\phi ::= \Psi|\phi \;\&\&\; \phi \mid \phi \;\;||\; \phi \mid !\phi \mid X\phi \mid F\phi \mid G\phi \mid \phi\, U \phi$$

AP is the grammar element for atomic propositions, which are formulated over the Petri net elements. AP are defined as boolean expressions over the enabling of transitions and the marking of the places.

$$AP ::= true \mid false \mid deadlock \mid initial \mid en(T') \mid \Theta \bowtie \Theta$$
$$\Theta ::= n \mid \#p \mid bounds(P') \mid -\Theta \mid \Theta \circ \Theta$$



ID: recIRQ1

0 Label: ...

2.0

Magnets: Center only

Tags:

Type: Exponential

Ok: object selected.

100 %

Figure 3 A screenshot of the GreatSPN interface when drawing the Petri net model for the Mutual Exclusion Problem. Full-size DOI: 10.7717/peerj-cs.823/fig-3

where $n \in \mathbb{N}$, $p \in P$, $P' \subseteq P$, $T' \subseteq T$, $\bowtie \in \{==, <, <=, >, >=\}$ is a comparison operator, and $\circ \in \{+, -, *, /\}$ is an arithmetic operator. The state formula *deadlock* evaluates to true for all states that do not enable any transition; *initial* is true in the initial state of the system; *en* (T') is satisfied in all states enabling at least one transition $t \in T'$; $\#p$ evaluates to the cardinality of place p in the current marking; $bounds(P')$ is the maximum sum of token counts of all places in P' in every reachable marking.

Figure 3 is a screen-shot of the GreatSPN GUI while editing a Petri net model. The model represents the abstract behaviour for a mutual exclusion algorithm in which two processes[1] may either stay out of the critical section at will (place NonCS$_i$ and self-loop with transition StayNonCS$_i$) or may receive an interrupt (firing of transiton recIRQ$_i$) to then proceed to request access to the mutually exclusive resource (place TryCS$_i$).

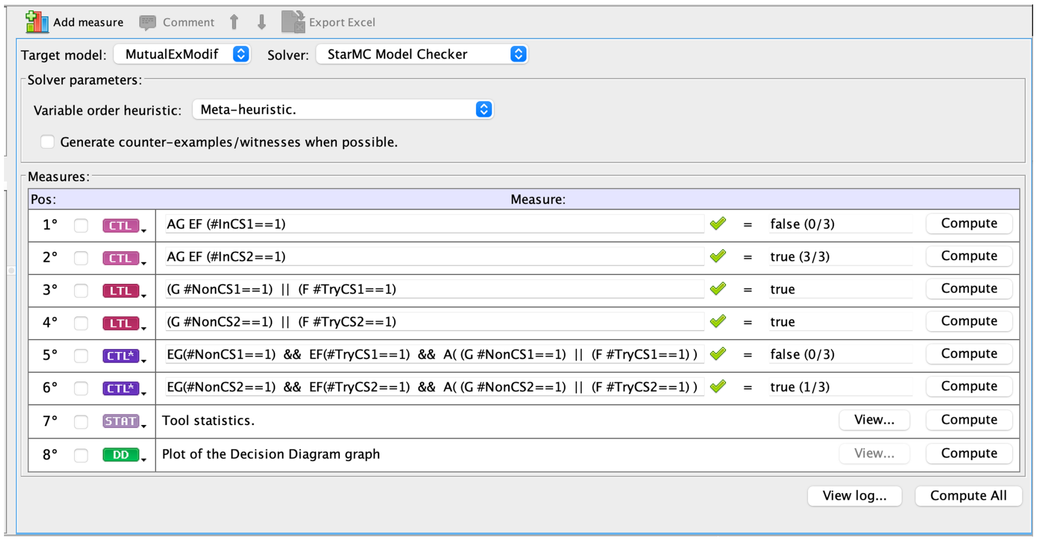
The relevance of this example to CTL* is pointed out very clearly in the work (*Emerson & Halpern, 1986*) of Emerson and Halpern, that proposes the following verification steps. If we want to specify that the mutual exclusion system should have both behaviours (either process i never tries to access, or at a certain point it will try to access), we may write the quantified LTL formula:

$Prop1 : A(G(\#\mathrm{NotCS}_i == 1) \,||\, (F\, \#\mathrm{TryCS}_i == 1))$

The model does indeed satisfy the specification of *Prop1*. Note that the formula is satisfied also if no process is allowed to stay forever in place #NotCS$_i$, given that on all paths it is possible to try to access the critical section. If we want instead to be more specific and add the more stringent requirement that both behaviours should be present (even if on different paths), we can write:

$Prop2 : EG(\#\mathrm{NotCS}_i == 1)\ \&\&\ EF(\#\mathrm{TryCS}_i == 1)\ \&\&\ A(G(\#\mathrm{NotCS}_i == 1) \,||\, (F\, \#\mathrm{TryCS}_i == 1))$

[1] For simplicity we have used a P/T model, so the model is not parametric in the number of processes. To draw a parametric model, colored Petri nets can be used from the same interface, and a P/T model can be obtained by automatic unfolding.



Target model: MutualExModif Solver: StarMC Model Checker

Solver parameters:

Variable order heuristic: Meta-heuristic.

Generate counter-examples/witnesses when possible.

Measures:

Pos:			Measure:				
1°		CTL	AG EF (#InCS1==1)	✓	=	false (0/3)	Compute
2°		CTL	AG EF (#InCS2==1)	✓	=	true (3/3)	Compute
3°		LTL	(G #NonCS1==1) || (F #TryCS1==1)	✓	=	true	Compute
4°		LTL	(G #NonCS2==1) || (F #TryCS2==1)	✓	=	true	Compute
5°		CTL*	EG(#NonCS1==1) && EF(#TryCS1==1) && A((G #NonCS1==1) || (F #TryCS1==1))	✓	=	false (0/3)	Compute
6°		CTL*	EG(#NonCS2==1) && EF(#TryCS2==1) && A((G #NonCS2==1) || (F #TryCS2==1))	✓	=	true (1/3)	Compute
7°		STAT	Tool statistics.			View...	Compute
8°		DD	Plot of the Decision Diagram graph			View...	Compute

Figure 4 **A screenshot of the interface that allows to insert queries for the `starMC` model checker.**
Full-size DOI: 10.7717/peerj-cs.823/fig-4

Prop2 is not an LTL formula, and no equivalent LTL formula exists, as proved in *Emerson & Halpern (1986)*. *Prop2* is not a CTL formula either since the third term has no equivalent CTL. *Prop2* is also true, but if we modify the net model to include a singular behaviour for process 1, so that it never receives an interrupt (modelled by removing transition recIRQ$_1$) then for process 2 both properties are true, but for process 1 *Prop1* holds while *Prop2* does not. So *Prop1* (an LTL formula) is not able to discriminate (and therefore to specify) the modified behaviour.

`starMC` is available both as a command line tool or from inside the GUI. Figure 4 shows a screenshot of the query interface of the `GreatSPN` GUI when checking the above propositions for the modified mutual exclusion model. The model checker receives a list of queries to be verified, with their language type (LTL, CTL, CTL*) explicitly specified. LTL formulae are implicitly quantified as "forall paths". The queries tagged as STAT and DD provide statistics and a pdf graphical representation of the DD of the RS. The queries tagged CTL are checked with the standard CTL model checker of GreatSPN (*Amparore, Beccuti & Donatelli, 2014*), while the queries tagged as LTL and CTL* are checked with `starMC`. Once a query is computed, the GUI shows if it holds in the initial state (true/false), as well as the cardinality of the sat-set (together with the RS size). For the modified mutual exclusion model being examined, $|RS| = 3$.

Indeed *Prop1* is true for both processes, while *Prop2* is false for the first process and true for the second one, showing the higher discriminating power of CTL*.

STARMC: THE LOGICAL VIEW

Background: LTL model checking

It is well-known, as in *Vardi (1995)* or *Baier & Katoen (2008*, p. 429), that for every LTL formula $A\varphi$ it is possible to generate a Generalized Büchi Automaton that accepts all and only the paths that satisfy ϕ, so we shall define them next.

Algorithm 1 **Check LTL $m_{init} \models A\varphi$ as: $\forall \pi \in \mathscr{P}^{\infty}_{M}(m_{init})$: $\pi \models \varphi$.**

```
1: procedure CHECK∀LTL($M$, $\varphi$)
2:     $\varphi' \leftarrow \neg\varphi$
3:     $\mathscr{A} \leftarrow$ translate $\varphi'$ into a Generalized Büchi Automaton
4:     $\mathscr{M} \leftarrow$ BA of the Kripke model $M(RG)$
5:     $\mathscr{M} \otimes \mathscr{A} \leftarrow$ synchronized product of $\mathscr{M}$ and $\mathscr{A}$
6:     if $\mathscr{L}(\mathscr{M} \otimes \mathscr{A})$ is empty then return true
7:     else return false
8:     end if
9: end procedure
```

Definition 6 (GBA). *A Generalized Büchi Automaton (GBA) is a tuple $\mathscr{A} = \langle Q, AP, \delta, Q_0, \mathscr{F} \rangle$, where Q is a finite set of locations, AP is a set of atomic proposition labels, $\delta \subseteq Q \times AP \times Q$ is a total transition relation, $Q_0 \subseteq Q$ is the set of initial locations, and $\mathscr{F} = \{F_i \mid F_i \subseteq Q\}_{i=1}^{n}$ is the set of accepting sets of locations.*

A GBA in which there is a single accepting set ($|\mathscr{F}| = 1$) is called a Büchi automaton (BA tout-court).

The language $\mathscr{L}(\mathscr{A})$ of a GBA $\mathscr{A}$ is the set of infinite words on AP recognized by a run of $\mathscr{A}$ that visits infinitely often at least one location in each $F_i \in \mathscr{F}$.

The standard automata-theoretic approach for model checking LTL formulae over a Kripke model M, following the schema of *Vardi & Wolper (1986, 1994)* is outlined in Algorithm 1, that checks if the quantified LTL formula $A\phi$ is satisfied for all paths π starting in the initial state m_0. To do so it builds the GBA $\mathscr{A}$ of $\neg\phi$ (line 3), the BA $\mathscr{M}$ of the Kripke model M (line 4), and the synchronized product GBA $\mathscr{M} \otimes \mathscr{A}$ (line 5), whose language is the intersection of the languages of M and $\mathscr{A}$. If this language is empty, no path π satisfies $\neg\phi$ and therefore ϕ holds on all paths $\pi \in \mathscr{P}^{\infty}_{M}(m)$.

When dealing with Petri nets, the model to be checked is a stuttered RG, which is translated into the Kripke model $M(RG)$ according to Definition 2. The corresponding automaton $\mathscr{M}$ is then built (line 4). Note that Büchi automata recognize words over edges, while Kripke models define a language of sequence of states (and associated atomic propositions). The transformation takes as accepting set $\mathscr{F} = \{RS\}$ and moves the labelling function of a state to all the incoming arcs, which requires a translation, based on *Vardi & Wolper (1986)*, for the initial state (that has no incoming arc) that consists in adding a fictitious initial state s_{pre}, leading to the following definition.

Definition 7 (BA of a Kripke model). *The Büchi automaton $\mathscr{M} = \langle Q, AP, \delta, Q_0, \mathscr{F} \rangle$ of a Kripke model $M = \langle S, E, L \rangle$, with $L : S \rightarrow 2^{AP}$, for the initial states $S_0 \subseteq S$, is defined as follows: $Q = S \cup \{s_{pre}\}$, with $s_{pre} \notin S$; $Q_0 = \{s_{pre}\}$; $\forall s, s' \in Q \setminus s_{pre}$: $\delta(s, s', l) \in \delta$ iff $(s,s') \in E$ and $l = L(s')$; $\forall s' \in S_0$, $\delta(s_{pre}, s', l) \in \delta$ iff $l = L(s')$; $\mathscr{F} = \{S\}$.*

Figure 5A shows the Büchi automaton built for the Kripke model in Fig. 1D, for $S_0 = s_1$.

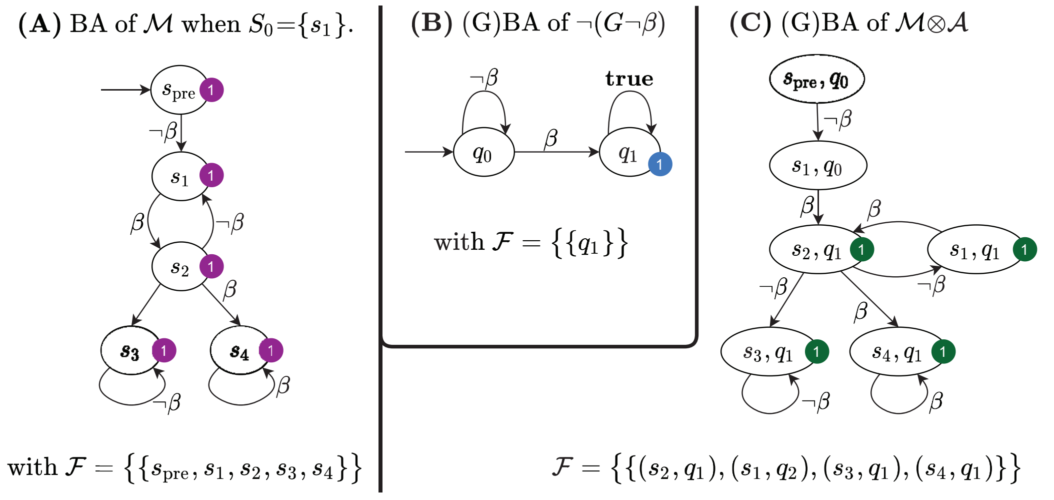

(A) BA of $\mathcal{M}$ when $S_0=\{s_1\}$.
s_{pre}
$\neg\beta$
s_1
β
$\neg\beta$
s_2
β
s_3
s_4
$\neg\beta$
β
with $\mathcal{F} = \{\{s_{pre}, s_1, s_2, s_3, s_4\}\}$
(B) (G)BA of $\neg(G\neg\beta)$
$\neg\beta$
true
q_0
β
q_1
with $\mathcal{F} = \{\{q_1\}\}$
(C) (G)BA of $\mathcal{M}\otimes\mathcal{A}$
s_{pre}, q_0
s_1, q_0
s_2, q_1
s_1, q_1
s_3, q_1
s_4, q_1
$\mathcal{F} = \{\{(s_2, q_1), (s_1, q_2), (s_3, q_1), (s_4, q_1)\}\}$

Figure 5 **The Büchi automata built by the LTL model checking procedure in Algorithm 1.**
Full-size DOI: 10.7717/peerj-cs.823/fig-5

We now define a simplified synchronized product of Büchi automata, for the case in which one automaton is a GBA and the other is a BA derived from a Kripke model (therefore with a single acceptance set that includes all states).

Definition 8 (Synchronized product GBA $\mathscr{M} \otimes \mathscr{A}$). *Given a BA $\mathscr{M} = \langle Q^M, AP, \delta^M, Q_0^M, \mathscr{F}^M = \{Q^M\}\rangle$ and GBA $\mathscr{A} = \langle Q^A, AP, \delta^A, Q_0^A, \mathscr{F}^{\mathscr{A}}\rangle$, their synchronized product is the GBA $\mathscr{M} \otimes \mathscr{A} = \langle Q, AP, \delta, Q_0, \mathscr{F}\rangle$ where*

$$
\begin{aligned}
&Q = Q^M \times Q^A\\
&Q_0 = Q_0^M \times Q_0^A\\
&\mathscr{F} = \{F_i\}_{i=1}^n : \forall q^A \in F_i^A, \forall q^M \in Q^M : (q^A, q^M) \in F_i\\
&\delta = \subseteq Q \times AP \times Q : ((q_i^A, q_i^M), a, (q_j^A, q_j^M)) \in \delta \text{ iff}\\
&\qquad\qquad (q_i^A, a, q_j^A) \in \delta^A \text{ and } (q_i^M, a, q_j^M) \in \delta^M
\end{aligned}
$$

It is known (*e.g.* *Baier & Katoen, 2008*, p. 156) that if $\mathscr{A} = \mathscr{A}_1 \otimes \mathscr{A}_2$, then $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}_1) \cap \mathscr{L}(\mathscr{A}_2)$. The Büchi automaton for the LTL formula $\neg\Phi$ with $\Phi = G \neg \beta$ is shown in Fig. 5B; its synchronized product with the Büchi automaton in Fig. 5A is shown in Fig. 5C. The product BA has always a single initial state (s_{pre}, q_0). The set of accepting subsets $\mathscr{F}$ for the example is $\mathscr{F} = \{F_1\}$, with $F_1 = \{(s, q_1), \forall s \in RS\}$. Since, indeed, there is an infinite path that starts in (s_{pre}, q_0) and that visits infinitely often at least one state in F_1, then $\mathscr{L}$ ($\mathscr{M} \otimes \mathscr{A}$) is not empty and therefore *the initial state of the net* does not satisfy the LTL formula $G \neg \beta$. Note that the initial state of the net is the successor of the initial state of $\mathscr{M} \otimes \mathscr{A}$.

From LTL to SAT∃LTL

A limitation of Algorithm 1 is that it only computes if the initial state m_{init} satisfies the formula $A\varphi$. For CTL*, we shall need instead the entire sat-set. Moreover, we switch to existentially-quantified LTL formulae, since they are easy to compute in symbolic form,

Algorithm 2 **Sat-set of a quantified LTL formula $E\varphi$.**

```
1: procedure SAT∃LTL(M, $\varphi$)
2:     $\mathscr{A} \leftarrow$ GBA of $\varphi$
3:     $\mathscr{M} \leftarrow$ BA of Kripke model $M(RG)$, with $S_0 = RS$
4:     $\mathscr{M} \otimes \mathscr{A} \leftarrow$ synchronized product of $\mathscr{M}$ with $\mathscr{A}$.
5:     $\mathscr{F}$ is the set of the acceptance sets of $\mathscr{M} \otimes \mathscr{A}$
6:     $Z_0 \longleftarrow succ(Q_0^{\mathscr{M}\otimes\mathscr{A}}, \delta^{\mathscr{M}\otimes\mathscr{A}})$
7:     $M(\mathscr{M} \otimes \mathscr{A})$ is the Kripke model of $\mathscr{M} \otimes \mathscr{A}$
8:     switch $type(\mathscr{A})$ do
9:         case $weak$:
10:            $AS \leftarrow$ SATCTL($M(\mathscr{M} \otimes \mathscr{A})$, $EF\ EG\ F_1$)
11:        case $terminal$:
12:            $AS \leftarrow$ SATCTL($M(\mathscr{M} \otimes \mathscr{A})$, $EF\ F_1$)
13:        case otherwise
14:            $AS \leftarrow$ SATE$_{\text{FAIR}}$G($M(\mathscr{M} \otimes \mathscr{A})$, $true$, fair = $\mathscr{F}$):
15:    return map $(AS \cap Z_0)$ over RS
16: end procedure
```

noting that a state $(s \models A\varphi) \equiv (s \models \neg E\ \neg\varphi)$. In this section we describe the extension of Algorithm 1 for sat-set computation as SAT∃LTL(M, φ).

A remarkable result of *Clarke, Grumberg & Hamaguchi (1997)*, Theorem 2, p. 54) shows that LTL model checking can be actually performed using a *fair CTL* model checker, not on the Kripke structure $\mathscr{M}$ but on the product GBA $\mathscr{M} \otimes \mathscr{A}$. In particular, only a single CTL operator, $E_{\text{fair}}G\ true$, is needed to check arbitrary LTL formulae.

We briefly recall the definitions of CTL fairness. A *fairness condition* F is a subset of states that must be visited infinitely often. A path $\pi \in \mathscr{P}_M^\infty(s)$ satisfies a *fairness condition* F if the states of F appear infinitely often on π. A *fairness set* $\mathscr{F}$ is a set of *fairness condition* F, that must all be met together. A path π is a *fair path* w.r.t. $\mathscr{F}$ iff π statisfies all fairness conditions of $\mathscr{F}$. A state s satisfies the *fair* CTL formula $E\phi$ w.r.t. fairness set $\mathscr{F}$ iff (1) it satisfies the CTL formula $E\phi$, and (2) $\exists\pi \in \mathscr{P}_M^\infty(s)$ that is a fair path w.r.t. $\mathscr{F}$.

A state s satisfies a quantified LTL formula $E\varphi$ if in the product GBA $\mathscr{M} \otimes \mathscr{A}$ there is a path from s, q_0 that visits infinitely often the states in the acceptance sets $\mathscr{F}$, *i.e.* the same definition of the *fairness condition* acceptance in fair CTL. Thus if we consider the GBA as a Kripke structure, finding the states that satisfy $E\varphi$ becomes equivalent to finding the locations that originate fair paths, *i.e.* finding the states on the equivalent Kripke structure that satisfy $E_{\text{fair}}G\ true$ w.r.t. the fairness set $\mathscr{F}$. The definition of SAT∃LTL(M, φ) that performs LTL model checking using fair CTL is described in Algorithm 2.

The procedure first translates the path formula φ into a GBA $\mathscr{A}$ (line 2), it then builds the BA $\mathscr{M}$ of the input model M (line 3), described by its RG. Note that, since the goal is the sat-set computation, we need to consider all states as possible initial states, by taking $S_0 = RS$ in definition 7. Figures 6A and 6B show the BA $\mathscr{M}$ built from the RG of

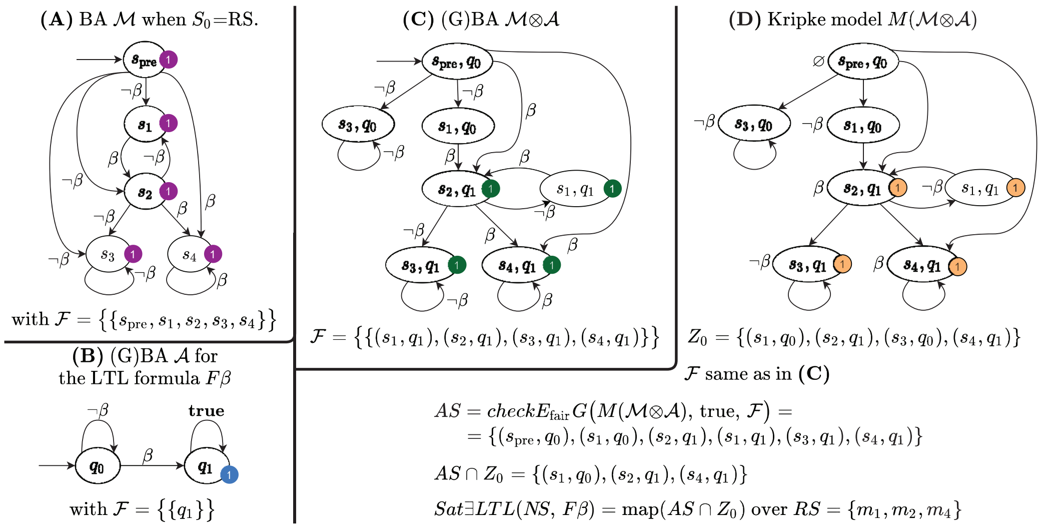

(A) BA $\mathcal{M}$ when S_0=RS.
with $\mathcal{F} = \{\{s_{pre}, s_1, s_2, s_3, s_4\}\}$
(B) (G)BA $\mathcal{A}$ for the LTL formula $F\beta$
with $\mathcal{F} = \{\{q_1\}\}$
(C) (G)BA $\mathcal{M}\otimes\mathcal{A}$
$\mathcal{F} = \{\{(s_1,q_1),(s_2,q_1),(s_3,q_1),(s_4,q_1)\}\}$
(D) Kripke model $M(\mathcal{M}\otimes\mathcal{A})$
$Z_0 = \{(s_1,q_0),(s_2,q_1),(s_3,q_0),(s_4,q_1)\}$
$\mathcal{F}$ same as in (C)
$AS = checkE_{fair}G(M(\mathcal{M}\otimes\mathcal{A}), \text{true}, \mathcal{F}) = \{(s_{pre},q_0),(s_1,q_0),(s_2,q_1),(s_1,q_1),(s_3,q_1),(s_4,q_1)\}$
$AS \cap Z_0 = \{(s_1,q_0),(s_2,q_1),(s_4,q_1)\}$
$Sat\exists LTL(NS, F\beta) = \text{map}(AS \cap Z_0)$ over $RS = \{m_1, m_2, m_4\}$

Figure 6 **The Büchi automata built by Algorithm 2 for the sat-set computation of ELTL formulae.**
Full-size DOI: 10.7717/peerj-cs.823/fig-6

the net of the previous example, and the BA $\mathscr{A}$ for the formula $F\beta$. The algorithm then builds (line 4) the GBA $\mathscr{M} \otimes \mathscr{A}$, as in Definition 8. The $\mathscr{M} \otimes \mathscr{A}$ automaton is then translated into the Kripke model $M(\mathscr{M} \otimes \mathscr{A})$ (line 6 and 7). This translation is trivial due to the particular structure of $\mathscr{M} \otimes \mathscr{A}$, since all arcs that enter a state carry the same set of atomic propositions, that can then be associated to the state itself. The sat-set of the CTL formula $E_{\text{fair}}G$ *true* for $M(\mathscr{M} \otimes \mathscr{A})$ is then computed (lines 8 to 14), which is then mapped back to the states of the initial model (the RS of the Petri net) in line 15.

The $\mathscr{M} \otimes \mathscr{A}$ of the example is shown in Fig. 6C: there is a single initial state (s_{pre}, q_0) and the accepting set is $\mathscr{F} = \{F_1\}$, where $F_1 = \{(s,q_1), \forall s \in RS\}$, therefore, in this case, $\mathscr{M} \otimes \mathscr{A}$ is a BA. The Kripke model $M(\mathscr{M} \otimes \mathscr{A})$ of the synchronized product $\mathscr{M} \otimes \mathscr{A}$ is shown in Fig. 6D, together with all the sets computed by Algorithm 2.

Some aspects of this procedure need a deeper explanation, in particular the three different ways of computing the sat-set (lines 8–14) and the Z_0 construction and its use (lines 6 and 15).

[Lines 8–14] This part computes the set of states that originate an infinite path accepted by the $\mathscr{M} \otimes \mathscr{A}$ BA. Line 14 is the general case: the results in *Burch et al. (1992)* and *Clarke, Grumberg & Hamaguchi (1997)* show how to compute the sat-set through the model-checking of the fair CTL formula $E_{\text{fair}}G(\mathit{true}, \mathit{fair} = \mathscr{F})$ on $\mathscr{M} \otimes \mathscr{A}$. The procedure for the sat-set computation of $E_{\text{fair}}G(\mathit{true}, \mathit{fair} = \mathscr{F})$ for a Kripke model is reported in Algorithm 3. Line 10 and 12 correspond to two specific cases, as it was shown (*Bloem, Ravi & Somenzi, 1999*) that if $\mathscr{M} \otimes \mathscr{A}$ is a *weak* or *terminal* BA, the computation of $E_{\text{fair}}G$ *true* can be simplified.

A BA is *weak* if (1) it is possible to find a partition $\{Q_i\}$ of its set of locations Q so that each Q_i is either contained in the accepting set or it is disjoint from it and (2) the $\{Q_i\}$ are partially ordered s.t. the transitions of the automaton never move from Q_i to Q_j unless $Q_i \leq Q_j$. In this case the only way for a path to visit infinitely often a state of the accepting

Algorithm 3 **Emerson-Lei algorithm for $E_{\mathrm{fair}}G\psi$ on Kripke model M and fairness constraints $\mathscr{F}$.**

```
1: procedure SatEfairG($M$, $\psi$, $\mathscr{F}$)
2:   $S \leftarrow Sat(\psi)$
3:   repeat
4:     $S' \leftarrow S$
5:     for each $F_i \in \mathscr{F}$: do
6:       $Y \leftarrow$ SatCTL($M$, $E\ (S\ U\ (F_i \cap S))$)
7:       $S \leftarrow S \cap$ SatCTL($M$, $EX\ Y$)
8:     end for
9:   until $S = S'$ // Repeat until fixed point is reached.
10:  return $S$
11: end procedure
```

set F_1 is to eventually be confined inside one $Q_i \subseteq F_1$, and an accepting run is a witness for the CTL formula $EF\ EG\ F_1$. A *terminal* BA is a weak BA in which the $\{Q_i\}$ are maximal elements in the partial order. In this case an accepting run is a witness for the CTL formula $EF\ F_1$. The time complexity for model checking the two CTL formulae is linear in the size of the model, while the cost for the general case in line 14 is instead quadratic. Note that Algorithm 2 tests the weak and terminal condition on $\mathscr{A}$, and not on the (usually) much larger $\mathscr{M} \otimes \mathscr{A}$, as it was proven in *Bloem, Ravi & Somenzi (1999)* that if $\mathscr{A}$ is weak (terminal) so is $\mathscr{M} \otimes \mathscr{A}$. When this is the case the simplified procedure obviously applies also to $M(\mathscr{M} \otimes \mathscr{A})$, as we do.

[Lines 6 and 15]. Z_0 computed in line 6 is the set of immediate successor of the initial state(s) of the $\mathscr{M} \otimes \mathscr{A}$. This step is needed because of the fictitious s_{pre} state introduced in $\mathscr{M}$ by the translation of Definition 7. The map operation in line 15 maps pairs (s_i, q_j) onto the marking of s_i.

The sat-set of the $EF\beta$ for the Petri net system of the example is also listed in Fig. 6D. Note that $(s_3, q_1) \models E_{\mathrm{fair}}G\ true$, as there is indeed an infinite loop on the (s_3, q_1) state, which is an accepting state. But s_3 corresponds to the Petri net marking m_3, a deadlock state that does not satisfies β, so certainly $s_3 \not\models EF\beta$. Indeed $(s_3, q_1) \notin Z_0$ since the witness path of $E_{\mathrm{fair}}G\ true$ for (s_3, q_1) corresponds to a path in the automata of the formula that does not start from the initial location q_0.

The work in *Emerson & Lei (1987)* gives a symbolic algorithm (known as *Emerson-Lei algorithm*) for model checking fair CTL properties, while (*Burch et al., 1992*) provides its fixed point characterization. Its time complexity is quadratic in the size of the automaton being checked. Algorithm 3 is an high-level view of the this fixed point characterization for the sat-set computation of $E_{\mathrm{fair}}G\psi$ formulae for a Kripke model M, with a set of fairness constraints $\mathscr{F}$. In the algorithm, the call to SatCTL(M, Φ) returns the sat-set of the CTL formula Φ (without fairness constraints) on the Kripke model M. The Until and next path operators at line 6 and 7 use sets instead of atomic propositions or state formulae, for simplicity.

Algorithm 4 **CTL* model-checking algorithm.**

```
1: procedure SatCTL*(M, Ψ)
2:   switch Ψ do
3:       case $a \in AP$: return $\{s \in S^M \mid s \models a\}$
4:       case $\Psi_1 \wedge \Psi_2$:
5:           return SatCTL*($\Psi_1$) ∩ SatCTL*($\Psi_2$)
6:       case $\neg\Psi$: return $S^M \setminus$ SatCTL*(Ψ)
7:       case $E\phi$: return Sat∃LTL(M, rewrite($\phi$))
8: end procedure
```

CTL* model-checking procedure

To understand the CTL* model checking procedure of `starMC`, we have to explain how a single CTL* formula is divided into multiple quantified LTL formulae that can be checked using Sat∃LTL only. It may be worth starting from an example. Let's consider the CTL* formula

$$\Psi = EFGE(\alpha\, U(AX\beta))$$

where α, β are atomic propositions, and define the following (sub-)formulae:

$$\Psi_1 = AX\beta \qquad \Psi_2 = E(\alpha U(AX\beta))$$
$$\Psi_3 = E(\alpha\, U a_{\Psi_1}) \qquad \Psi_4 = EFGa_{\Psi_3}$$

Let $Sat(\Psi)$ be the sat-set of Ψ. Assume that we have a model checking procedure SatLTL(Ψ) that computes $Sat(\Psi)$ if Ψ is a quantified LTL formula. Similarly, assume that a procedure SatCTL(Ψ) computes $Sat(\Psi)$ if Ψ is a CTL formula. Formulae Ψ_1 and Ψ_3 can be considered as both quantified LTL and CTL formulae, while Ψ_2 is not LTL, and Ψ_4 is not CTL. If we assume that $a\Psi_i$ is an atomic proposition that holds in state s iff $s \models \Psi_i$, then we can compute $Sat(\Psi)$ using only SatLTL on the sequence of formulae: Ψ_1, Ψ_3 and Ψ_4. *Vice-versa*, $Sat(\Psi)$ cannot be computed using only SatCTL, since SatCTL(Ψ_2) allows to rewrite Ψ into Ψ_4 which is an LTL formula for which it is well-known that no equivalent CTL one exists. According to *Emerson & Lei (1987)*, the idea of substitution was already present in *Emerson & Sistla (1984)* and it was later used in *Visser & Barringer (2000)*.

Algorithm 4 is the logical view of the CTL* model-checking procedure implemented in `starMC`. The algorithm follows the ideas in *Emerson & Lei (1987)*, and builds a CTL* model-checker based on Sat∃LTL. It contains one case per type of state formula (by definition a CTL* formula is a state formula). The only non-trivial case of Algorithm 4 is $E\varphi$ in line 7: if φ is an LTL path formula, Sat∃LTL can be directly applied to the formula, if instead φ contains additional quantifications, we first have to rewrite φ as an LTL path formula (by substituting quantified sub-formulae with newly created atomic propositions). Both cases are treated in a unified manner through the call of Sat∃LTL (rewrite(φ)). Each formula identified for the substitution is a Maximal Proper Quantified

Sub-formula (MPQS). A MPQS is defined in terms of the so-called Maximal Proper State Sub-formula (MPSS); see (Baier & Katoen, 2008), def.6.86.

Definition 9 (Maximal Proper State Sub-formula). *State formula Ψ is a MPSS of ρ whenever Ψ is a sub-formula of ρ that differs from ρ and that is not contained in any other proper state sub-formula of ρ.*

Let MPSS(ρ) be the set of all maximal proper state sub-formula of ρ.

Example (from (Baier & Katoen, 2008)). If $\rho = E\varphi$ and

$$\varphi = X(AGEFa) \wedge FGE(Xa \wedge Gb)$$

then MPSS(φ) $= \{AGEFa,\ E(Xa \wedge Gb)\}$. Back to the initial example, $MPSS(\Psi) = \{\Psi_2\}$ and $MPSS(\Psi_2) = \{\Psi_1\}$.

Let's consider this second example:

$$\Psi = EFG\,\underbrace{E(\alpha\,U\,\underbrace{(\alpha \wedge \underbrace{AX\beta}_{\Psi_3})}_{\Psi_2})}_{\Psi_1}$$

than Ψ_2 is a MPSS of Ψ_1 and Ψ_3 is a MPSS of Ψ_2. In our model-checking procedure we consider instead a refinement of the MPSS definition, that we call Maximal Proper Quantified Sub-formula (MPQS). The motivations for introducing MPQS with respect to previous work and MPSS is given in "MPSS *vs.* MPQS".

Definition 10 (MPQS). *A Maximal Proper Quantified Sub-formula (MPQS) Ψ of a formula ρ is a MPSS of ρ which is a quantified state formula, i.e. either $\Psi = E\phi$ or $\Psi = A\phi$.*

The MPQS decomposition maximises the length of path formulae whenever possible. The MPQS decomposition of the example CTL* formula is:

$$\Psi = EFG\,\underbrace{E(\alpha\,U(\alpha \wedge \underbrace{AX\beta}_{\Psi_3}))}_{\Psi_1}$$

The state sub-formula Ψ_2 cannot be a MPQS of Ψ_1, since Ψ_2 is not a quantified state formula.

REWRITE is the procedure to identify MPQS; its logical view is described in Algorithm 5, which is based on the syntactical recognition of the input formula ρ. Operators and atomic propositions are left unchanged (lines 3–9), when $\rho = E\phi$ (line 10) the formula ρ is rewritten as the atomic proposition a_ρ and to appropriately assign a_ρ, the sat-set of ρ is computed through the indirect recursive call to SATCTL $^*(\rho)$.

PREVIOUS WORK

While the main sources of inspiration for the CTL* model checking procedure presented in "StarMC: The Logical View" have already been introduced, in this section we discuss and compare with previous work with a broader view. We shall first discuss the different

Algorithm 5 **Identification and replacement of MPQSs.**

```
1: procedure REWRITE($\rho$)
2:     switch $\rho$ do
3:         case $\rho_1 \wedge \rho_2$:
4:             return REWRITE($\rho_1$) “$\wedge$” REWRITE($\rho_2$)
5:         case $\neg\rho$: return “$\neg$” REWRITE($\rho$)
6:         case $X\rho$: return “$X$” REWRITE($\rho$)
7:         case $\rho_1 \; U \; \rho_2$:
8:             return REWRITE($\rho_1$) “$U$” REWRITE($\rho_2$)
9:         case $a \in AP$: return “$a$”
10:        case $E\phi$:
11:            // $\rho = E\phi$ is a MPQS. Replace $\rho$ with an AP $a_\rho$.
12:            $AP \leftarrow AP \cup \{$new $a_\rho\}$
13:            $Sat \leftarrow$ SATCTL*($\rho$)
14:            for each $s \in Sat$ do
15:                $L(s) \leftarrow L(s) \cup \{a_\rho\}$
16:            end for
17:            return “$a_\rho$”
18: end procedure
```

approaches to CTL* model-checking and report on past and existing CTL* model-checkers. Since the CTL* model-checking procedure of `starMC` is based on LTL model-checking, we also provide a summary of the main approaches to LTL model-checking.

CTL* model-checking approaches

CTL* model checking procedures can be distinguished depending on whether they are conducted on the model itself or on an extended model (as in case of the synchronized product).

Model-checking the model itself

An example of this approach is when the CTL* formula is first translated into μ-calculus and then the μ-calculus model-checking procedure is applied. This can be just a straightforward implementation of the fixed-point definition of the full μ-calculus or the more efficient algorithm proposed in *Emerson & Lei (1986)* for μ_2-calculus, based on the observation that the formula translated from CTL* belong to the μ_2 fragment of the μ-calculus. The efficacy of *symbolic* techniques for model checking of μ-calculus was shown in *Burch et al. (1992)*. In this approach the whole CTL* formula is translated into a single μ-calculus property: this translation is exponential in the size of the formula (*Bhat & Cleaveland, 1996*). A linear translation from of CTL* to *first order* μ-calculus, a form of μ-calculus with data, is proposed in *Cranen, Groote & Reniers (2011)*. Note that, in this case, a CTL* formula may not produce a single first-order μ-calculus expression.

Model-checking an extended model

The model is extended to incorporate the formula information. The extended model is therefore constructed ad-hoc for each formula being checked. This approach may be further refined according to how the formula structure is encoded/exploited.

1. *Recursive descendent approaches based on formula substitution*. As in the original papers by *Emerson & Lei (1985*, p. 90) and *Emerson & Lei (1987)*, sub-formulae are identified and substituted, thus reducing the sat-set computation for CTL* to a sequence of sat-set computations for LTL. The complexity of this CTL* model-checking procedure is of the same order as that of LTL. It is well-known (*Baier & Katoen, 2008*) that for CTL*, as for LTL, the model-checking problem is PSPACE-complete, with a cost linear in the size of the system and exponential in the size of the formula (because of the Tableau or the Büchi automaton construction).
2. *Hesitating alternating tree automata (HAA)*. A different approach is presented in *Visser & Barringer (2000)*, where CTL* formulae are translated into HAA. It is mentioned that to build the HAA for the CTL* formula, when an $\exists\varphi$ sub-formula is encountered it is necessary to build the Büchi automata that accepts all infinite words recognized by φ. The synchronized product of the HAA of the formula with the Kripke model is again an HAA, and checking that the original formula is satisfied on the Kripke model corresponds to checking that the language of the synchronized product HAA is empty. Also this approach is based on the idea of MPSS (or MPQS) substitution with a label. Although described in a different manner, it seems that the approach based on HAA shares many similarities with the approach described in "StarMC: The Logical View". It basically does in an implicit manner what the SATCTL* in Algorithm 4 does in an explicit one. In the same paper the authors also propose a model checking procedure based on games, where the objective is to find a winning strategy for the non-emptiness game on the synchronized product of the Kripke model and the HAA of the formula (Theorem 2 in *Visser & Barringer (2000)*).

Sat-set computation for $E_{\text{fair}}G\,\Psi$

The work in *Emerson & Lei (1986)*, already mentioned for its contribution in proving that μ_2-calculus can be efficiently model-checked, also provides a succinct translation from fair-CTL to μ_2-calculus, paving the way to efficient model checking of fair-CTL, in particular for the sat-set computation of $E_{\text{fair}}G\Psi$. The algorithm is provided in an abstract form, regardless of the specific data structures choice. It is readily suitable for a symbolic implementation, which is what we have used for `starMC`.

A different symbolic solution for the computation of a fair cycle is provided by the OWCTY algorithm (*Černá & Pelánek, 2003*), which does not compute sat-sets but aims at building counter-examples as quickly as possible. A comparison of the two approaches for LTL model checking can be found in *Duret-Lutz et al. (2011)*.

Sat-set computation for ELTL

Buchi-based model checking of LTL of a formula Φ is defined in *Vardi & Wolper (1986)*, as recalled in "StarMC: The Logical View". The paper in *Burch et al. (1992)* shows a symbolic model checking procedure for LTL. It is based on an implicit tableau construction and on its symbolic representation. It is then shown that the sat-set of an LTL formula can be reduced to the fair CTL symbolic model-checking for the μ-calculus expansion of EG true, under appropriate fairness constraints.

A later work (*Clarke, Grumberg & Hamaguchi, 1997*) encodes the tableau of the LTL formula along with the model variables in a single SMV (*Clarke et al., 1996*) model, and explicitly reduces the LTL model checking to a fair-CTL formula, using the existing and unmodified CTL model checker of SMV. Model checking fair CTL is based on the Emerson-Lei algorithm (*Emerson & Lei, 1986*). This procedure, together with its implementation in BDD is well explained in the survey paper by *Rozier (2011)*. The survey also cover many other related topics, including a comparison of the expressiveness of LTL, CTL and CTL* both in theory and in practical applications.

`starMC` exploits the Buchi-based approach in *Vardi & Wolper (1986)*, that makes the synchronized product of the Büchi automaton and the model, and uses the algorithm in *Emerson & Lei (1986)* for the detection of the accepting runs and the consequent construction of the set of states that satisfy the formula. The library `Spot` (*Duret-Lutz, 2014*; *Duret-Lutz et al., 2016*) provides different, efficient, ways to build a Büchi automaton from an LTL formula. The generated automaton may have state-based or transition-based acceptance. This latter choice, based on the translation defined in *Couvreur (1999)*, implemented in Spot using decision diagrams, is usually the most efficient. `starMC` nevertheless uses state-based automata, since it allows an easier re-use of the SatCTL module of `GreatSPN`.

A number of possible optimizations have been presented in the literature for LTL model-checking. For example in *Duret-Lutz et al. (2011)* a modified algorithm called Self-Loop Aggregation Product (SLAP) is proposed, where instead of building a synchronized product in an extended space (model and automaton), the joint state is kept into separate aggregates. Accepting runs are then searched only in those aggregates that may contain cyclic runs over the accepting states. For the time being starMC does not include this kind of algorithmic optimization, although a certain attention has been taken to avoid inefficiencies in the symbolic implementation.

We do not review here the numerous techniques for the on-the-fly model-checking of LTL, since these techniques concentrate on establishing the truth value of a formula in the initial state, while the Sat∃LTL procedure needed for CTL* model checking requires the construction of the sat-sets of the LTL sub-formulae.

CTL* model checkers

According to a relatively old paper (*Barringer et al., 2002*) there was a CTL* model-checker available in the Rainbow verification system (*Barringer et al., 2002*): the tool is not available any longer and we could find no published details on how the model checker was built, only that it was based on HAA. Previous papers by the same authors evaluated whether it was the case to enhance the tool SPIN with CTL* (*Visser & Barringer, 1998*; *Visser & Barringer, 2000*), but apparently it was never done and/or reported in the literature.

There is an implementation of μ-calculus for the nuXmv tool (*Cavada et al., 2014*), which, according to its website, could lead to a CTL* model-checker but, for the time being, only CTL* formulae that are either LTL or CTL are actually processed.

The `LTSmin` tool (*Kant et al., 2015*) includes a CTL* model checker. It allows to model check a Petri net by translating the CTL* formula to be checked into an equivalent μ-calculus one, using the procedures defined in *Dam (1990)*. A model checker of μ-calculus for Petri nets is available also in TINA (*Berthomieu, Ribet & Vernadat, 2004*), but no translator from CTL* to μ-calculus is provided.

MPSS *vs.* MPQS

In the literature there have been slightly different definitions of the CTL* sub-formulae to be substituted and, although they are all correct and adequate for CTL* model-checking based on SAT∃LTL, different definitions may lead to a different number and/or to a different structure of the Büchi automata being built. The work in *Emerson & Lei (1987)* does not contain a definition in strict sense of the formulae to be substituted, but it states that, for each formula $E\phi$, where ϕ is a path formula, then the set of formulae to be substituted by atomic propositions are the "top-level proper existential sub-formulae Eq_i of ϕ that are not sub-formulae of any other sub-formula Er of $E\phi$, where Er is different from $E\phi$ and from Eq_i". The Eq_i are therefore the largest quantified sub-formulae of the formula ϕ.

In later times the substitution of formulae with atomic propositions was based on the definition of MPSS (*Visser & Barringer, 2000*), which is equivalent to the one given by def.6.86 in *Baier & Katoen (2008)*, reported in this paper as Definition 9. The definition of MPSS encounters two problems when deriving an algorithm, it may re-label more formulae than strictly necessary, and it may result in a non-deterministic algorithm since the CTL* grammar is ambiguous. Indeed in the considered example:

$$\Psi = EFG\,\underbrace{E(\alpha\,U\,\underbrace{(\alpha \wedge \underbrace{AX\beta}_{\Psi_3})}_{\Psi_2})}_{\Psi_1}$$

Since ψ_2 is both a state and a path formula due to the ambiguity of the CTL* language, there are two evaluation strategies:

1. Evaluate Ψ_1 as $E\ (\alpha U a\psi_2)$;
2. Evaluate Ψ_1 as $E\ (\alpha U(\alpha \wedge a\psi_3))$;

Both strategies are correct, but they are not equivalent. The former strategy is based on the identification of MPSS sub-formulae. While correct, it inhibits possible reductions and optimizations for Büchi Automata generation, since the label $a\psi_2$ clearly carries less information than the $(\alpha \wedge a\psi_3)$ formula. This example also shows that MPSS are different from "top-level proper existential sub-formulae" which are at the base of the CTL* model checking algorithm in Emerson & Lei (1987) and that we have formally defined as MPQS in Definition 10. The MPQS decomposition of the example CTL* formula is:

$$\Psi = EFG\,\underbrace{E(\alpha\,U(\alpha \wedge \underbrace{AX\beta}_{\Psi_3}))}_{\Psi_1}$$

The state sub-formula ψ_2 cannot be a MPQS of Ψ_1, since Ψ_2 is not a quantified state formula.

STARMC: THE INNER VIEW

This section provides an insight on the implementation of `starMC`, which consists in a DD-based implementation on the algorithms of "StarMC: The Logical View". The implementation relies on the Meddly and Spot libraries, and on the symbolic CTL model checker already available in `GreatSPN`.

In `starMC` DDs are used to encode all data structures, but for the BA of the (sub-) formulae, that are instead represented in explicit form. Petri net state spaces, Kripke models and Büchi automata are all expressed in DD forms using the M×D of the NSF and the MDD of the states (when needed). Most computations of the model checker work with DDs that encode "potential" states. Potential states are a superset of RS that arise because the NSF, which is built before the RS, may include the firing of transitions also from states that are not reachable for the given initial marking. Algorithms (like state space generation) that work forward, from the initial marking onward, never generate unreachable states, but this is not the case in the implementation of some path operators, that works backwards (*e.g.* the neXt). Working on potential states is a standard choice for DD-based model-checkers, and intersections with the RS are done only when strictly needed.

CTL* model checking

Implementation of SATCTL* *and* REWRITE

We do not provide a detailed implementation view of Algorithm 4 for the sat-set computation for CTL*, but we simply identify the data structures used. SAT∃LTL returns an MDD, RS in line 6 is an MDD and the set operations in line 5 and 6 are also implemented as MDD operations. The value returned by REWRITE (ϕ) is an LTL expression following Definition 4. An implementation choice worth noticing is whether S^M in line 3 and 6 of the algorithm refers to the MDD of the actual RS or of the set of potential states (*PS*). This part is common to the CTL model-checker of `GreatSPN`, for which the default choice is to manipulate potential states. Intersection with reachable states is done only for negation (line 6) (set difference from RS) and for the sat-set of `true`, for which RS is used. Atomic propositions (line 3) are also defined on the PS.

Algorithm 6 Synchronous product construction between $\mathscr{M} = \langle RS^M, NSF^M \rangle$ and $\mathscr{A} = \langle Q, AP, \delta, Q_0, \mathscr{F} \rangle$.

```
1: procedure BuildSynchProduct($M$, $\mathscr{A}$)
2:     // Build the MDD of the initial states.
3:     $Z_0 \leftarrow$ MDD()
4:     for each location $q_0 \in Q_0$: do
5:         for each edge $e = q_0 \xrightarrow{a} q'$ in $\delta$: do
6:             $Z_0 \leftarrow Z_0 \cup AddLoc(Sat^M(a), q')$
7:         end for
8:     end for
9:     // Build the Next State Function M×D
10:    $NSF \leftarrow$ M×D()
11:    for each edge $e = q \xrightarrow{a} q'$ in $\delta$: do
12:        $nsf^e \leftarrow AddLocX(NSF^M \cap (PS^M \times Sat^M(a)), q, q')$
13:        $NSF \leftarrow NSF \cup nsf^e$
14:    end for
15:    // $S_{\mathscr{F}}$: array of MDDs, one entry per accept. set $F_i \in \mathscr{F}$
16:    for each accepting set $F_i \in \mathscr{F}$: do
17:        $\mathrm{MDD}_{F_i} \leftarrow \cup\{edgeForVar(q) \mid \text{for each } q \in F_i\}$
18:        $S_{\mathscr{F}}[i] \leftarrow \{\mathrm{MDD}_{F_i}\}$
19:    end for
20:    return $\langle Z_0, NSF, S_{\mathscr{F}} \rangle$
21: end procedure
```

Implementation of $\mathscr{M} \otimes \mathscr{A}$ construction

The generation of the decision diagrams that encode the $\mathscr{M} \otimes \mathscr{A}$ automaton is summarized in Algorithm 6. The MDD (M×D) of $\mathscr{M} \otimes \mathscr{A}$ is defined to have $n + 1$ ($2n + 2$ resp.) levels: one level each for the n places of the net and one for the location of $\mathscr{A}$. We indicate with $\langle m, q \rangle$ a joint state of $\mathscr{M} \otimes \mathscr{A}$, where m is a model state and q is an automaton location.

The algorithm builds the MDD and the M×D of $\mathscr{M} \otimes \mathscr{A}$ taking as input a graph description of the formula automaton $\mathscr{A}$ and the M×D for the NSF of $\mathscr{M}$. There is no MDD for the initial states of $\mathscr{M}$ since all states are considered as initial ones. The algorithm implements at the same time the translation of $\mathscr{M}$ into a Büchi automaton (Definition 7) and the $\mathscr{M} \otimes \mathscr{A}$ construction (Definition 8). The algorithm directly builds the Kripke model, without passing through the $\mathscr{M} \otimes \mathscr{A}$ BA as described in "StarMC: The Logical View". The DD representation of the $\mathscr{M} \otimes \mathscr{A}$ automaton of Algorithm 6 consists of three elements: M×D *NSF* for the δ function; an array $S_{\mathscr{F}}$ of MDDs of the accepting sets; and the MDD for the set Z_0 of states as defined by Algorithm 2.

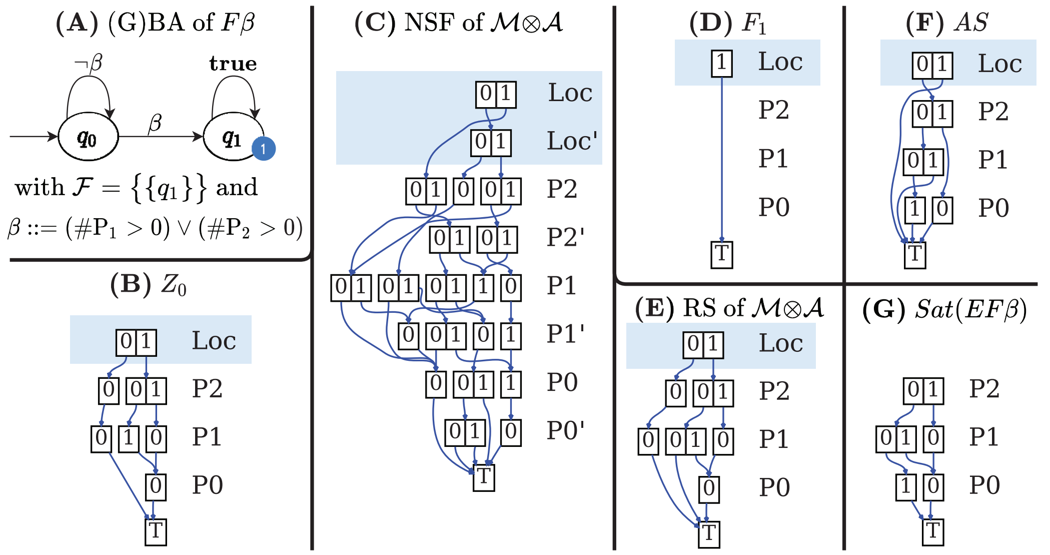

(A) (G)BA of $F\beta$
$\neg\beta$
true
β
q_0
q_1
1
with $\mathcal{F} = \{\{q_1\}\}$ and
$\beta ::= (\#P_1 > 0) \vee (\#P_2 > 0)$
(B) Z_0
(C) NSF of $\mathcal{M}\otimes\mathcal{A}$
(D) F_1
(E) RS of $\mathcal{M}\otimes\mathcal{A}$
(F) AS
(G) $Sat(EF\beta)$
Loc
Loc'
P2
P2'
P1
P1'
P0
P0'
T

Figure 7 **(A–G) A Büchi automaton of a CTL* formula and the DDs generated by its evaluation for the net in Fig. 1A.** Full-size DOI: 10.7717/peerj-cs.823/fig-7

Figures 7B–7G shows the DDs built by Algorithm 6 for the net of Fig. 1A and for the formula expressed by the Büchi automaton in Fig. 7A. These DDs represent the symbolic encoding of the information (models and sat-sets) of Fig. 6.

Construction of Z_0 (lines 4–8)

If q_0 is an initial state of $\mathscr{A}$ with an outgoing arc $(q_0 \xrightarrow{a} q')$ then Z_0 should include all states $\langle s, q' \rangle$ for all states s that satisfy a ($s \in Sat(a)$). The function $AddLoc(d, q)$ takes a MDD d and sets the location level to q. Figure 7B shows the MDD of Z_0.

Construction of the NSF (lines 9–14)

The M×D for NSF is built by modifying the M×D NSF^M of the input model $\mathscr{M}$ to include only "appropriate states" of $\mathscr{M}$ and to add the locations of $\mathscr{A}$. For each edge $q \xrightarrow{a} q'$ of $\mathscr{A}$ a new M×D is created (line 12), by modifying the NSF of $\mathscr{M}$ to reach only $Sat^M(a)$ markings (the states of $\mathscr{M}$ that satisfies a), and at the same time by moving the GBA location from q to q'. Function $AddLocX(v, q, q')$ takes a M×D v and adds a relation $q \to q'$ for the first primed and unprimed levels (automata locations). The transition relation δ of $\mathscr{M} \otimes \mathscr{A}$ is encoded as a NSF from the union of all the edge M×Ds (line 13). Figure 7C shows the M×D of the NSF of the $\mathscr{M} \otimes \mathscr{A}$ of the example.

Identification of the array of accepting subsets $S_{\mathscr{F}}$ (lines 15–19)

The set $S_{\mathscr{F}}$ is simply implemented by the array $S_{\mathscr{F}}$ of MDDs, one entry per F_i. According to Definition 8, each accepting subsets F_i is the Cartesian product of F_i^A the i-th subset of final locations of $\mathscr{A}$ with all the states of $\mathscr{M}$. The Cartesian product F_i can be realized by adding one level on top of an MDD v that encodes the states of $\mathscr{M}$. The node of this top level has a down-arrow to v for each location $q \in F_i^A$. A more efficient solution is actually implemented in Algorithm 6, based on the observation that we can consider "potential" states of $\mathscr{M}$ by considering any combinations of tokens in any place. The corresponding

Algorithm 7 Implementation view of SAT∃LTL.

```
1: procedure SAT∃LTL($M$, $\varphi$)
2:     $\mathscr{A} \leftarrow$ translate $\varphi$ into a Büchi Automaton
3:     $\langle Z_0, NSF, S_{\mathscr{F}} \rangle \leftarrow$ BUILDSYNCHPRODUCT($M$, $\mathscr{A}$)
4:     $S \leftarrow Saturate(Z_0, NSF)$
5:     $K = (S, NSF)$ is a Kripke model
6:     $AS \leftarrow MDD()$
7:     switch $type(\mathscr{A})$ do
8:         case $weak$:
9:             $AS \leftarrow$ SATCTL$(K, EF\ EG(S_{\mathscr{F}}[1]))$
10:        case $terminal$:
11:            $AS \leftarrow$ SATCTL$(K, EF(S_{\mathscr{F}}[1]))$
12:        case otherwise:
13:            $AS \leftarrow$ SATE$_{\text{FAIR}}$G$(K, \mathbf{true}, fair = S_{\mathscr{F}})$:
14:    $Sat(\exists\varphi) \leftarrow RemLoc(AS \cap Z_0)$
15:    return $Sat(\exists\varphi)$
16: end procedure
```

MDD, in fully reduced form, is very compact: it has $n + 1$ levels, the node of the top level carries the F_i^A elements, all other levels are skipped. This MDD can be built by taking (line 17) the union of MDDs created by the Meddly function *edgeForVar*$(q, n + 1)$, which creates a fully reduced MDD in which level $n + 1$ is set to q and all other levels are skipped.

In the automaton of the example $\mathscr{F} = \{F_1\}$, with $F_1 = \{q_1\}$, therefore $S_{\mathscr{F}}$ is an array with a single element, the MDD that combines the value 1 (encoding of q_1) for the location level with any combination of tokens in all places. The MDD is depicted, in fully reduced form, in Fig. 7D.

Implementation of SAT∃LTL

SAT∃LTL computes the set of states of M that satisfy the quantified LTL formula $E\varphi$. Algorithm 7 provides an implementation view of Algorithm 2. Line 2 is a call to Spot for formula φ, that builds a Büchi automaton in textual HOA form. The DDs of $\mathscr{M} \otimes \mathscr{A}$ are built by the call in line 3. The set of reachable states S in $\mathscr{M} \otimes \mathscr{A}$ is then generated in line 4 using *saturation* (*Ciardo, Lüttgen & Siminiceanu, 2001*).

Figure 7A is the automaton built by Spot for the formula $F\beta$, to be evaluated for the net in Fig. 2A, with $\beta = (\#P1 > 0 \text{ or } \#P2 > 0)\}$. Figures 7B–7D are the three DDs returned by the call of BUILDSYNCHPRODUCT in line 3, while Fig. 7E is the result of the Saturation procedure on $\mathscr{M} \otimes \mathscr{A}$.

Note that in the $\mathscr{M} \otimes \mathscr{A}$ construction of Algorithm 6 all informations on edge labelling is lost, since it is now irrelevant, and, to find the set of accepting states (AS) of the formulae in lines 9, 11, and 13, we can consider $\langle S, NSF \rangle$ as a Kripke model (line 5). Lines 7 to 13 follow the same logic of Algorithm 2.

As for Algorithm 2 the computed AS set is not the accepting set of the original formula: the operations in line 14 build the MDD intersection of the MDDs of AS and Z_0 and applies to the resulting MDD the *RemLoc* function, that removes the DD level that stores the location index, thus obtaining an MDD that encodes the reachable states of the Petri net model that satisfy $E\varphi$.

The MDD of the AS set for the example is given in Fig. 7F. It has been computed by the call to the CTL model-checker in line 11, since $\mathscr{A}$ is a terminal automaton. Note that DD represents AS in the potential state space. Moreover not all the accepting states in AS are in the sat-set of $E\varphi$, either because they are not reachable (are in $PR \backslash RS$) or because they are not "aligned" with the initial conditions of $\mathscr{A}$, otherwise said, they have to belong to Z_0. The result of the intersection and *RemLoc* operations in line 14 is shown in Fig. 7G, that encodes the 3 satisfying markings (all markings of RS except the one where all places have zero tokens). Note that $\mathscr{M} \otimes \mathscr{A}$ states $(q_0, 1, 1, 0)$ and $(q_1, 0, 0, 0)$, are in the DD of the AS but nor $(1, 1, 0)$ nor $(0, 0, 0)$ are in the DD of $Sat(EF\beta)$. The first one is a state which is not in the RS of $\mathscr{M} \otimes \mathscr{A}$, the second one is not in Z_0. Both are removed by the intersection in line 14.

Implementation of $E_{fair}G$

The logical view for the sat-set computation of $E_{\mathrm{fair}}G$ given in Algorithm 3 is quite close to its DD-based implementation. The $E_{\mathrm{fair}}G$ is called by Algorithm 7 on a Kripke model encoded with an M×D for its NSF, a MDD for its states, and an array of MDDs for the set of fairness constraints $\mathscr{F}$. All DDs are for $n + 1$ variables, the n places of the Petri net model and the location of the automaton of the formula. The SATCTL calls in line 6 and 7 of Algorithm 3 have been implemented with the pre-existing DD-based CTL model checker of `GreatSPN`, modified so as to treat $n + 1$ levels. The intersections in lines 6 and 7 are straightforward MDD intersections.

TESTING RESULTS

We have tested the *correctness* of the results and the *performance* of the tool. All tests were performed using the publicly available (mcc.lip6.fr/2019/archives/2019-mcc-models.tar.gz) MCC2019 benchmark (*Kordon et al., 2019*). The MCC2019 benchmark includes 1,018 model instances, that are parametric variations (in structure or in the initial marking) of 94 distinct colored and P/T models. There are both academic (*e.g.* Philosophers, Kanban, Erathostenes, *etc.*) as well as relevant industrial models (ARM processor cache, biochemical networks, UML models, CAN bus models, *etc.*). For each model instance different properties are defined: basic Petri net properties (like cardinality of the state space, absence of deadlock, boundedness and liveness), 32 LTL properties and 32 CTL properties, for a total of about 60 k LTL and CTL formulae. The benchmark includes all known formula evaluations (truth value in the initial marking), making MCC data a valuable benchmark for (Petri net) tools. The tests use `Spot` version 2.9.6, the latest `Meddly` (tag `7a31ca8` on gitHub), `LTSmin` version 3.0.2. and `GreatSPN` (tag `4439bde`).

The semantics of LTL and CTL is defined over infinite paths, but Petri net models with deadlocks also feature finite paths. For MCC, CTL is directly defined on RG and not

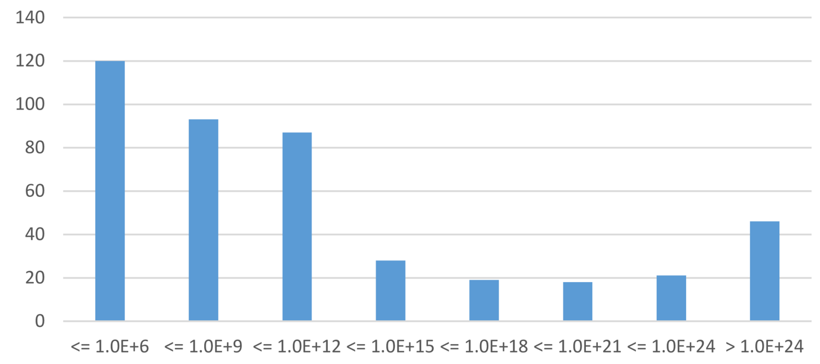


Figure 8 Distribution of the state space size of the 434 model instances built by `starMC` in less than 60 s. Full-size DOI: 10.7717/peerj-cs.823/fig-8

on Kripke structures, hence the successor function may not be total. In that case, the CTL semantics is axiomatically defined (*Lichtenstein, Pnueli & Zuck, 1985*) for the set S_d of deadlock states as: $EX\ S_d = \emptyset$, and $EG\ S_d = S_d$. LTL and CTL* properties are interpreted considering a proper Kripke structure (*i.e.* stuttered) built on the RG according to Def. 2.

The benchmark does not include any CTL* property for the MCC models (and for any other models we could find). Therefore we have algorithmically generated a set of CTL* formulae from the MCC *CTLCardinality* queries by randomly deleting path quantifiers with a probability of 70%. The top-most quantifiers are always kept, in order to preserve consistency with the CTL* grammar. The sat-set cardinalities of the LTL and CTL formulae are also missing in the benchmark. We have therefore conducted the comparison tests on the cardinalities of the produced sat-sets by checking consistency between multiple tools.

Resource setting. All experiments have been performed on dedicated cpu cores (Intel Xeon CPU E5-2630 v3 at 2.40 GHz) with a limit of 2 GB of memory and a 60 s timeout for state space generation and another 60 s to check each formula. Variable orders were pre-computed using the state-of-the-art heuristic of *Amparore, Donatelli & Ciardo (2020a)*.

Examined models. Within the 60 s limit, starMC is able to build the RS of 434 model instances, from 77 different models. This set includes many very large model instancess, both in terms of places and transitions (up to thousands) and number of states. Figure 8 shows a chart of the distribution of the sizes of the 434 model instances. Two of these instances have a state space of the order of 10^{478}, one in the order of 10^{603} (numbers not even representable in a double precision floating point). The 67 model instances with more than 10^{21} states (the two rightmost columns in Fig. 8) correspond to 29 different models.

Analysis' objectives. The characteristics that have been tested are:

CT: **C**orrect **T**ruth value of the formula for the initial state

CC: **C**orrect **C**ardinality of the sat-set of the formula

PE: **PE**rformance (execution times)

Table 1 Summary of the experiments with starMC and RGMeDD, on 13,664 CTL queries.

Characteristics	Value	Characteristics	Value
No. of queries	13,664	Same Time both	317
starMC terminates	10,223	Both timed-out	2,447
RGMeDD terminates	10,740	Only RGMeDD timed-out	476
Both terminate	9,747	Only starMC timed-out	993
starMC faster	938	starMC timeout and RGMeDD out of memory	1
RGMeDD faster	8,492	Mismatches in sat-set cardinality	0

CT test based on MCC results

starMC results are checked against the available truth values for CTL and LTL properties. For each instance we computed the result for 32 CTL and 32 LTL properties, for a total of 27,776 queries (434 * 64) queries. With the test resource limitations, starMC was able to terminate about 80% of the queries. All unanswered queries did not finish in the time bound. None of the query evaluation generated a not-enough-memory error. For 7 of these 434 models, the query format was corrupted due to bad namings. These 7 models were thus dropped. We found no mismatch w.r.t. the MCC published values for the remaining 427 models.

To get a better accuracy we need to verify more precise information like the sat-set cardinalities (the **CC** test), as in the next steps, but before proceeding to sat-set computation we have investigated the possible overhead caused by the (exponential) size of the Büchi automata. We have computed the size of the Büchi automata for about 14 k LTL queries. The largest automata produced by Spot has 70 locations and 356 edges. The average number of locations (edges) is 3.95 (7.7). This correspond to a maximum and average number of path operators equal to 12 and 4.7 respectively. Note that starMC may build more than one Büchi automata per formula when model-checking CTL and CTL* formulae, due to nesting.

CC and PE tests based on RGMeDD

This test compares the results of starMC and RGMeDD on CTL formulae. RGMeDD is the CTL model checker (*Amparore, Beccuti & Donatelli, 2014*) of GreatSPN, based on the standard recursive sat-set computation of CTL. The test therefore compares two different approaches to CTL model checking: a standard fixed point implementation (RGMeDD), and an automata-based implementation (starMC). Both tools use the same library for DD manipulation, moreover the same variable orders have been used. The benchmark is again on 427 instances, for a total of 13,664 CTL queries (427 * 32).

Table 1 summarizes the behaviour of the two tools, while the plots in Fig. 9 report the execution times in linear form Fig. 9A and log form Fig. 9B: dots below the diagonal are queries for which RGMeDD is faster than starMC and *vice-versa*. Executions that have timed out are marked as *TO*.

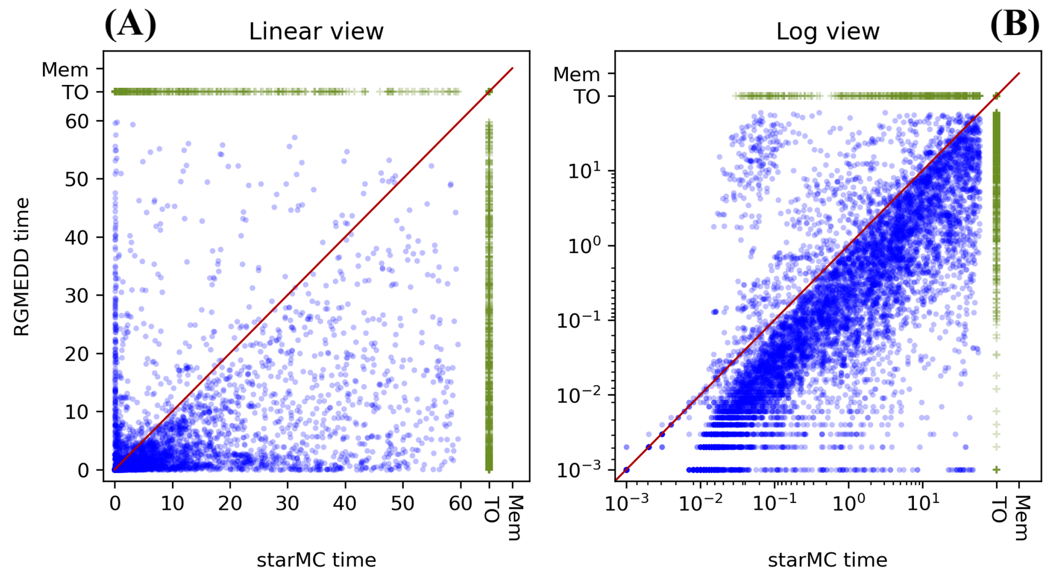


Figure 9 **(A–B) Execution times of `starMC` *vs.* `RGMeDD`, on 13,664 CTL queries.**
Full-size DOI: 10.7717/peerj-cs.823/fig-9

`RGMeDD` completes 10,740 queries (79%) from 427 different model instances, while `starMC` completes 10,223 queries (75%) from 399 different model instances. All the 9,747 queries completed by both tools (71%) produced sat-sets of equal cardinality (**CC** test).

We have also analyzed the different behaviour in terms of solved queries and model instances, with `RGMeDD` being able to solve more queries but especially for more diverse instances.

Moreover, on the queries completed by both tools (blue dots in the plot), `RGMeDD` was faster than `starMC` on 87% of the queries, slower on the 10% and equal time (up to millisec) on 3%. There are a number of cases in which the execution times are rather different (values close to only the x or the y axis). This result for the **PE** test is somehow unexpected: considering that the same DD library is used, and the same variable orders, the differences are possibly due to the different model-checking approaches, and we actually expected the standard CTL procedure of `RGMeDD` to be more efficient in almost all practical cases. Factors that could make `starMC` faster than `RGMeDD` are the optimization of the formula done by Spot when building the Büchi automaton and a different use of the potential state space in the two implementations.

Tests based on `LTSmin`

The last tool assessment is based on a comparison with `LTSmin` (*Kant et al., 2015*): **CC** and **PE** are assessed for LTL, CTL and CTL* formulae. `LTSmin` is run using the *pins2lts-sym* interface with the *–ctlstar* option, which first converts the input formula into μ-calculus and then applies the μ-calculus model-checker of `LTSmin`. It is worth noting that this translation may incur an exponential cost since `LTSmin` uses the translation described in *Dam (1990)*, which is in theory less efficient than using Büchi automata. `LTSmin` is also based on decision diagrams, provided by the multi-core library Sylvan (*van Dijk & van de*

Table 2 Summary of the experiments with starMC and LTSmin on CTL and LTL queries.

Characteristics	Value	Characteristics	Value
No. of queries	3,936 (123*32)	Same Time both	1
starMC terminates	2,990	Both timed-out	895
LTSmin terminates	1,758	Only LTSmin timed-out	1,277
Both terminates	1,708	Only starMC timed-out	50
starMC faster	485	Mismatches in sat-set cardinality	7
LTSmin faster	1,222		

Pol, 2017). To make a meaningful comparison, for each model instance we have enforced the use of the same variable order for the two tools. We have initially checked that both tools, when given the same variable order, produce exactly the same DD in output. Thus the choice of the variable order does not penalize one tool or the other, as long as it is the same for both tools. We give LTSmin the same time constraints as the other tools, *i.e.* 60 s to generate the state space, and 60 s to translate the formula in μ-calculus and perform the model checking. Since we have experienced inconsistencies of the LTSmin tool when dealing with models with deadlocks, we have considered deadlock-free models only. Moreover, only P/T models have been included in the analysis, since LTSmin does not treat colored models, leaving 222 model instances from 29 distinct models. For what concerns the type of queries, only LTL/CTL/CTL* queries of the "Cardinality" category have been checked, since LTSmin cannot express atomic propositions based on the enabling of transitions. We did not consider model instances for which none of the two tools was able to compute the state space, leaving 123 model instances from 28 different models. Of the 123 models, LTSmin builds the state space of 86 and starMC of 116. In all tests, LTSmin ran on four cores (to allow it to exploit the parallelism of the Sylvan library) and starMC on a single one, similar to the settings used for the MCC competition.

The first test (CC and PE for LTL and CTL formulae) compares the sat-set cardinalities and the execution times for LTL and CTL formulae, while the second test (CC and PE for CTL* formulae) extends the analysis to CTL* formulae, the final target of the whole testing procedure.

CC and PE for LTL and CTL formulae

We computed the sat-sets generated by starMC and LTSmin (running in *–ctlstar* mode) on LTL and CTL formulae on the 123 model instances selected as explained above, with 16 CTL formulae and 16 LTL ones for each model instance. The objective is to check the performance of the two model checkers (**PE**) and, for all queries solved by both tools, check that the same sat-sets' cardinalities are computed (**CC**). The results are summarized in Table 2, while a comparison of the execution times in log and linear form (as for the previous experiment) is reported in Fig. 10.

PE results: in the 60 s limits, starMC solves 50% more queries than LTSmin, although, in the subset of queries solved by both, LTSmin is faster than starMC on 2 out of 3 queries.

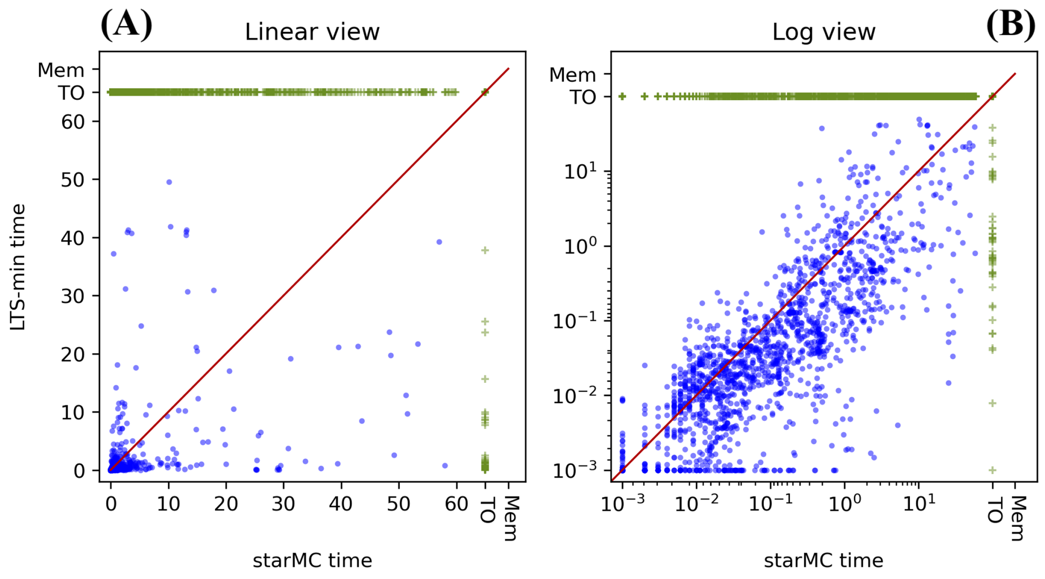


Figure 10 **(A–B) Execution times of `starMC` *vs.* `LTSmin` on CTL and LTL queries.**
Full-size DOI: 10.7717/peerj-cs.823/fig-10

Table 3 **Summary of the experiments with `starMC` and `LTSmin` on CTL* queries.**

Characteristics	Value	Characteristics	Value
No. of queries	1,968 (123*16)	`LTSmin` faster	536
`starMC` terminates	1,442	Both timed-out	505
`LTSmin` terminates	775	Only `LTSmin` timed-out	688
Both terminates	754	Only `starMC` timed-out	21
`starMC` faster	218	Mismatches in sat-set cardinality	10

CC results: of the 1,708 queries computed by both tools, we have a mismatch in the cardinalities of the sat-sets of 7 formulae (from different model instances). These are all LTL formulae. All results computed by `starMC` are consistent with the MCC known truth values. Since truth values of MCC are assigned according to majority of the output of the tools that participate in the competition, they may not be 100% reliable. We have therefore computed the sat-sets of the sub-formulae, which indicate that `LTSmin` is computing the wrong sat-sets, most likely due to a wrong translation to μ-calculus. For example, the sat-set of the formula named LTLCardinality-08 for the model BART-PT-002 has a structure "*XFGF*a", where a is an atomic proposition and the formula is implicitly quantified for all paths. `LTSmin` reports that the formula has an empty sat-set, although the sat-set of the formula "*FGF*a" is equal to the full state space, which clearly indicates contradictory results. The translation in μ-calculus of the former query leads to a recursive formula with 7 μ and 18 ν fixed point operators, and it is not trivial to assess whether there is an error in the translation of the formula into μ-calculus or in the model-checking procedure. This behaviour has been reported to the `LTSmin` developers.

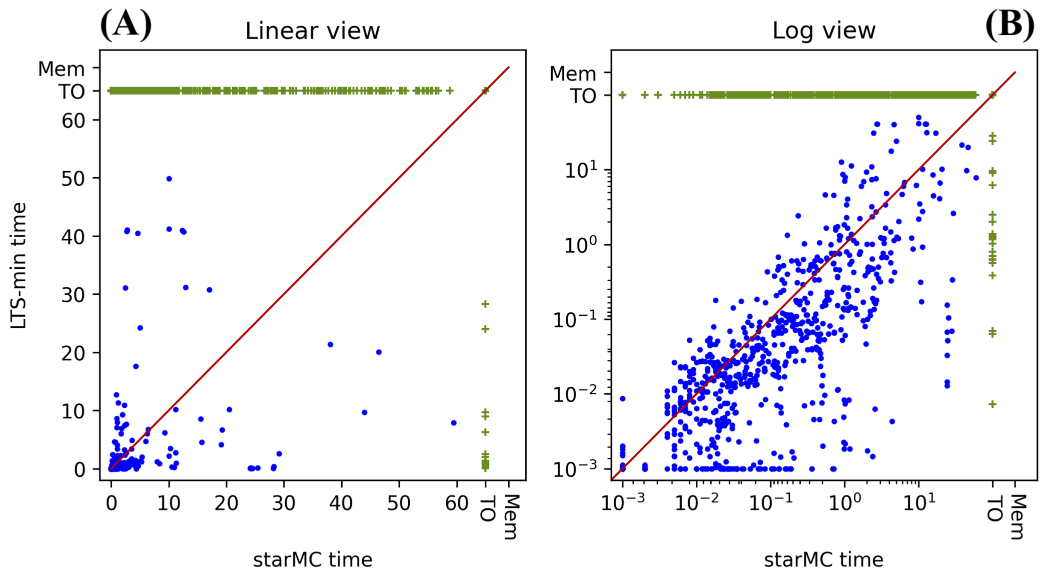


Figure 11 **(A–B) Execution times of `starMC` *vs.* `LTSmin` on CTL* queries.**
Full-size DOI: 10.7717/peerj-cs.823/fig-11

CC and PE for CTL formulae*

The tests of "CC and PE for LTL and CTL formulae" are here repeated for the set of generated CTL* formulae. The results are summarized in Table 3, while a comparison of the execution times in log and linear form (as for the previous experiments) is reported in Fig. 11.

PE: in the 60 s limits, `starMC` solves the highest number of queries (73%), almost twice those solved by `LTSmin`. In the subset of queries solved by both, `LTSmin` is faster than `starMC` on almost 4 out of 5 queries.

CC: of the 753 queries computed by both tools, we have a mismatch in the cardinalities of the sat-sets of 10 formulae (from different model instances). We have checked these formulae one by one, by computing the sat-set of sub-formulae, when meaningful and useful. In all but one case, the two tools differ radically: the sat-set being the whole state space for one and the empty set for the other, or *vice-versa*. In these cases it was not difficult to check that `LTSmin` is computing results that are inconsistent with the sat-sets of the sub-formulae computed by the tool itself. For the remaining case a special procedure was put in place that indicates that it is likely that `LTSmin` is computing the wrong result[2].

CONCLUSION

`starMC` is a CTL* model-checker that computes the set of reachable states that satify a CTL* formula. To the best of our knowledge it is the only available CTL* model-checker based on Büchi automata for Petri nets, and also the only available CTL* model-checker that does not require a translation into μ-calculus.

[2] There is a query of a model (named FMS-PT-00020 in MCC) for which the two sat-sets differ for a "small" portion: for `LTSmin` the sat-set is the whole RS, for `starMC` is a strict subset of RS (with a difference of 10^4 states out of the 10^{12} states in RS). Due to the complexity of model and formula we had to build a reduced model (applying standard Petri net reduction rules) with only 32 states, and a reduced formula, for which the two sat-sets differ by four states. In this case it was possible to check by hand that the 4 should not be in the sat-set, as computed by `starMC`.

starMC provides CTL* and LTL model-checking capabilities to the GreatSPN tool, improving the existing CTL model-checker. starMC is fully integrated into the GreatSPN GUI.

The implementation leverages two libraries: Spot for the translation from LTL sub-formulae to Büchi automata, and Meddly for decision diagram manipulation. The tool also includes the existing CTL model-checker of GreatSPN, modified to verify $\mathscr{M} \otimes \mathscr{A}$ structures.

Although many of the ideas behind the construction of starMC have been around for decades, there was not, as far as we know, a fully developed description on the algorithms for a (symbolic) implementation of a CTL* model-checker, that we consider to be a significant contribution of this paper. The availability of such a tool is also very important: for educational purposes both in university and in industries, and for application in real-life contexts. The testing section has shown that, even on reduced resources (2 GB of memory and execution time with a 60 s time-out) starMC can model-check very large state spaces (we reached 10^{480} states) for formulae with equivalent Büchi automata of more than 70 locations and 350 edges.

Petri nets only?

starMC has been developed for Petri nets, either in GreatSPN format or in the PNML standard format. Although having a model-checker fully integrated into a specification and verification tool is an advantage for the user, it may hamper its reuse in the research community. There are indeed tools like LTSmin that try to be as general as possible, by providing a (formalism agnostic) intermediate language. starMC exploits Petri nets to collect useful information: computation of place bounds (so the user does not have to identify them beforehand) and heuristics for variable orders, which are based on P-semiflows (a structural property of the Petri net). Both issues relates to performance and do not hamper the application of the results presented in this paper to other formalisms for DEDS; nevertheless the tool is not currently structured for direct reuse as a stand-alone multi formalism model-checker.

Tool’s limitations

starMC builds the whole state space beforehand even for properties that could be proved by inspection of a part of the state space (no on-the-fly verification), and it does not build counter-examples.

Answer to research questions

(**R1**): our testing campaign has shown that it is possible to realize an efficient and fully symbolic implementation of the computation of sat-sets of LTL and CTL* properties based on Büchi automata. (**R2**): the experiments conducted in the testing phase indicate that starMC can solve (significantly) more formulae than LTSmin, although, on the formulae solved by both tools, LTSmin is generally faster. (**R3**): an extensive comparison of different variable order heuristics has not been conducted yet, but the size of the system that we were able to solve show that, indeed, the variable order heuristic used, that was the best one for state-space exploration, as reported in *Amparore et al. (2019)*, is at least adequate for CTL* model-checking. (**R4**): the construction of counter-examples and witnesses has not been addressed in this paper.

Current and future work

`starMC` builds the whole state space beforehand even for properties that could be proved by inspection of a part of the state space (no on-the-fly verification). We are working on a model-checker that keeps the NSF in implicit form: the M×D of the NSF is substituted by a function that performs the firing of the Petri net transitions by directly manipulating the MDD of the state space being built. With a NSF in implicit form it is possible to implement an on-the-fly approach, in which the state space is built incrementally up to the point where the property can be (dis-)proven. The results could be extended to CTL* following the work in *Bhat, Cleaveland & Grumberg (1995)*, that develops an on-the-fly procedure for LTL and then extends it to CTL*.

No model-checker is fully useful if it does not produce meaningful counter-examples. `GreatSPN` already produces CTL counter-examples or witnesses for the initial state. Extending the current approach to CTL* may require a definite shift in the approach, for example by looking into the creation of evidence for μ-calculus (*Cranen, Luttik & Willemse, 2015*), while keeping an eye on the efficiency of the construction, as in *Jiang & Ciardo (2018)*.

Although CTL* allows to express fairness constraints in the formula, we plan to develop also a model checker for the fair variant of CTL. Although this may seem a straightforward task, considering the already existing implementation of $E_{\text{fair}}G\,\psi$, further analysis is needed to understand the most adequate form of fairness, in particular if we want to consider not only fairness based on visited states (and therefore among enabled actions), but also fairness of taken actions, to include, for example, the "Fairness running" clause of the NuSMV tool (*Cavada et al., 2014*).

The example in "CTL* Model-checking Procedure" points out that a CTL* model-checker could be realized based on a mixture of SAT∃LTL and SATCTL. Beyond the theoretical differences in complexity, the construction of such a CTL* model-checker requires more experiments to understand if there is a practical advantage in doing so. Our comparison in "CC and PE Tests Based on RGMeDD" points out that it could be so.

Availability

- A virtual machine, with the reproducible benchmark (scripts, models, tools, instructions) is available as a Zenodo permalink. The provided benchmark can be run either in quick or in full mode, which will take about 30 minutes/10 days, respectively, to complete. Instructions are provided inside the VM, available at: https://zenodo.org/record/5752419 with the name starMC-benchmark.ova.
- A second virtual machine with the starMC tool pre-installed can also be found at the same link, with the name starMC.ova.

- The source code of `GreatSPN` is available at https://github.com/greatspn/SOURCES.
- The benchmark data is available at https://github.com/amparore/starMC-benchmark.

ACKNOWLEDGEMENTS

We would like to thank Jaco van de Pol for the various insights given on LTSmin, the researchers and developers of the Spot and Meddly libraries, and all researchers across the world that have contributed to the MCC benchmark.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Elvio Gilberto Amparore conceived and designed the experiments, performed the experiments, analyzed the data, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Susanna Donatelli conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the paper, and approved the final draft.
- Francesco Gallà conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The virtual machine with the benchmark data and the virtual machine with the tools are available at Zenodo:

Elvio Gilberto Amparore, Susanna Donatelli, & Francesco Gallà. (2021). Replication Package (Virtual Machine) for article "starMC: an automata based CTL* model checker" [Data set]. In PeerJ Computer Science (preprint). Zenodo. https://doi.org/10.5281/zenodo.5752419. The provided benchmark can be run either in quick or in full mode, which will take about 30 min/10 days, respectively, to complete.

The source code of the tool is available at GitHub: https://github.com/greatspn/SOURCES.

The benchmark data is available at GitHub: https://github.com/amparore/starMC-benchmark.

REFERENCES

Aalst W. 1998. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers* **8**:21–66 DOI 10.1142/S0218126698000043.

Amparore E, Donatelli S. 2018a. Efficient model checking of the stochastic logic CSL^{TA}. *Performance Evaluation* **123–124**:1–34 DOI 10.1016/j.peva.2018.03.002.

Amparore EG. 2014. A new GreatSPN GUI for GSPN editing and CSL^{TA} model checking. In: *Proceeding of the 11th QEST Conference.* Vol. 8657. *LNCS*, Berlin: Springer, 170–173.

Amparore EG, Balbo G, Beccuti M, Donatelli S, Franceschinis G. 2016. 30 Years of GreatSPN. In: *Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi.* Berlin: Springer, 227–254.

Amparore EG, Beccuti M, Donatelli S. 2014. (Stochastic) model checking in GreatSPN. In: Ciardo G, Kindler E, eds. *35th International Conference on Application and Theory of Petri Nets and Concurrency, Tunis.* Cham: Springer, 354–363.

Amparore EG, Ciardo G, Donatelli S, Miner AS. 2019. i-Rank: a variable order metric for DEDS subject to linear invariants. In: *25th International Conference, TACAS 2019.* **11428**:Berlin: Springer, 285–302.

Amparore EG, Donatelli S. 2018b. GreatTeach: a tool for teaching (Stochastic) Petri Nets. In: *39th ATPN International Conference.* **10877**:Berlin: Springer, 416–425.

Amparore EG, Donatelli S, Ciardo G. 2020a. Variable order metrics for decision diagrams in system verification. *International Journal on Software Tools for Technology Transfer* **22(5)**:541–562 DOI 10.1007/s10009-019-00522-6.

Amparore EG, Donatelli S, Gallà F. 2020b. A CTL* model checker for Petri nets. In: Janicki R, Sidorova N, Chatain T, eds. *41st International Conference on Application and Theory of Petri Nets, PN2020, volume 12152 of Lecture Notes in Computer Science.* Vol. 12152. Berlin: Springer, 403–413.

Babar J, Miner A. 2010. Meddly: multi-terminal and edge-valued decision diagram library. In: *Proceedings of QEST Conference*. Piscataway: IEEE, 195–196.

Babiak T, Blahoudek F, Duret-Lutz A, Klein J, Kretnský J, Müller D, Parker D, Strejcek J. 2015. The Hanoi Omega-Automata format. In: *Computer Aided Verification-27th International Conference, CAV 2015, volume 9206 of Lecture Notes in Computer Science*. Berlin: Springer, 479–486.

Baier C, Katoen J. 2008. *Principles of model checking*. Cambridge: MIT Press.

Barringer H, Fellows D, Gough G, Williams A. 2002. Rainbow: development, simulation and analysis tools for asynchronous micropipeline hardware design. *The Computer Journal* **45(1)**:2–11 DOI 10.1093/comjnl/45.1.2.

Berthomieu B, Ribet P-O, Vernadat F. 2004. The tool TINA—construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research* **42**:2741–2756 DOI 10.1080/00207540410001705257.

Bhat G, Cleaveland R. 1996. Efficient model checking via the equational μ-calculus. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*. Washington, D.C.: IEEE Computer Society, 304.

Bhat G, Cleaveland R, Grumberg O. 1995. Efficient on-the-fly model checking for CTL*. In: *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*. 388–397.

Bloem R, Ravi K, Somenzi F. 1999. Efficient decision procedures for model checking of linear time logic properties. In: *International Conference on Computer Aided Verification, LNCS 1633*. Berlin: Springer, 222–235.

Burch J, Clarke E, McMillan K, Dill D, Hwang L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* **98(2)**:142–170 DOI 10.1016/0890-5401(92)90017-A.

Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S. 2014. The nuXmv symbolic model checker. In: *CAV International Conference*. Vol. 8559. *LNCS*, Berlin: Springer, 334–342.

Černá I, Pelánek R. 2003. Distributed explicit fair cycle detection (set based approach). In: *International SPIN Workshop on Model Checking of Software*. Berlin: Springer, 49–73.

Ciardo G, Lüttgen G, Siminiceanu R. 2001. Saturation: an efficient iteration strategy for symbolic state-space generation. In: *TACAS*. 328–342.

Clarke E, McMillan K, Campos S, Hartonas-Garmhausen V. 1996. Symbolic model checking. In: *International Conference on Computer Aided Verification*. Berlin: Springer, 419–422.

Clarke EM, Emerson EA. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logic of Programs, volume 131 of Lecture Notes in Computer Science*. Berlin: Springer, 52–71.

Clarke EM, Grumberg O, Hamaguchi K. 1997. Another look at LTL model checking. *Formal Methods in System Design* **10(1)**:47–71 DOI 10.1023/A:1008615614281.

Couvreur J-M. 1999. On-the-fly verification of linear temporal logic. In: Wing JM, Woodcock J, Davies J, eds. *FM'99—Formal Methods*. Berlin: Springer, 253–271.

Cranen S, Groote JF, Reniers MA. 2011. A linear translation from CTL* to the first-order modal μ-calculus. *Theoretical Computer Science* **412(28)**:3129–3139 DOI 10.1016/j.tcs.2011.02.034.

Cranen S, Luttik S, Willemse T. 2015. Evidence for fixpoint logic. In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015). 7–10 September 2015*. Berlin, Germany, 78–93.

Dam M. 1990. Translating CTL* into the modal μ-calculus. Technical Report ECS-LFCS-90-123, University of Edinburgh.

Dam M. 1994. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science* **126(1)**:77–96 DOI 10.1016/0304-3975(94)90269-0.

Donatelli S, Haddad S, Sproston J. 2009. Model checking timed and stochastic properties with CSL^{TA}. *IEEE Transactions on Software Engineering* **35(2)**:224–240 DOI 10.1109/TSE.2008.108.

Duret-Lutz A. 2014. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems* **5(1/2)**:31–54 DOI 10.1504/IJCCBS.2014.059594.

Duret-Lutz A, Klai K, Poitrenaud D, Thierry-Mieg Y. 2011. Self-loop aggregation product-A new hybrid approach to on-the-fly LTL model checking. In: *ATVA 2011. Proceedings, volume 6996 of Lecture Notes in Computer Science*. Berlin: Springer, 336–350.

Duret-Lutz A, Lewkowicz A, Fauchille A, Michaud T, Renault E, Xu L. 2016. Spot 2.0—a framework for LTL and ω-automata manipulation. In: *Proceedings of the 14th International Symposium of ATVA, volume 9938 of LNCS*. Berlin: Springer, 122–129.

Eisner C, Fisman D. 2016. *Chapter temporal logic made practical: handbook of model checking*. Berlin: Springer.

Emerson EA, Halpern JY. 1983. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic (preliminary report). In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 127–140.

Emerson EA, Halpern JY. 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM* **33(1)**:151–178 DOI 10.1145/4904.4999.

Emerson EA, Lei C. 1985. Modalities for model checking: branching time logic strikes back. In: *POPL '85*. 84–96.

Emerson EA, Lei C. 1986. Efficient model checking in fragments of the propositional μ-calculus (extended abstract). In: *LICS'86*. Washington, D.C.: IEEE Computer Society, 267–278.

Emerson EA, Lei C. 1987. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming* **8(3)**:275–306 DOI 10.1016/0167-6423(87)90036-0.

Emerson EA, Sistla AP. 1984. Deciding full branching time logic. *Information and Control* **61(3)**:175–201 DOI 10.1016/S0019-9958(84)80047-9.

Holzmann G. 2004. *The SPIN model checker: primer and reference manual*. Boston: Addison-Wesley.

Jiang C, Ciardo G. 2018. Generation of minimum tree-like witnesses for existential CTL. In: Beyer D, Huisman M, eds. *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, volume 10805 of Lecture Notes in Computer Science*. Berlin: Springer, 328–343.

Kant G, Laarman A, Meijer J, van de Pol J, Blom S, van Dijk T. 2015. LTSmin: high-performance language-independent model checking. In: *TACAS'2015, volume 9035 of LNCS*. Berlin: Springer, 692–707.

Koch I. 2019. Petri nets in systems biology: transition invariants, maximal common transition sets, transition clusters, Mauritius maps, and MonaLisa. In: *Carl Adam Petri: Ideas, Personality, Impact*. Berlin: Springer, 217–226.

Kondratyev A, Kishinevsky M, Taubin A, Cortadella J, Lavagno L. 1998. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems and Computers* **8(1)**:67–118 DOI 10.1142/S0218126698000055.

Kordon F, Berthomieu B, Ciardo G, Zilio SD, Gallà F, Hillah LM, Hulin-Hubard F, Jensen PG, Jezequel L, Amparore E, Le Botlan D, Liebke T, Meijer J, Miner A, Paviot-Adet E, Srba J, Thierry-Mieg Y, van Dijk T, Wolf K. 2019. Presentation of the 9th Edition of the Model Checking Contest. In: *TOOLympics, Proc. of TACAS, volume 11429 of LNCS*. Berlin: Springer, 50–68.

Kozen D. 1983. Results on the propositional μ-calculus. *Theoretical Computer Science* **27(3)**:333–354 DOI 10.1016/0304-3975(82)90125-6.

Lichtenstein O, Pnueli A, Zuck L. 1985. The glory of the past. In: *Logics of Programs*. Berlin: Springer, 196–218.

Murata T. 1989. Petri nets: properties, analysis and applications. *Proceedings of the IEEE* **77(4)**:541–580 DOI 10.1109/5.24143.

Pnueli A. 1977. The temporal logic of programs. In: *FOCS*. Washinton, D.C.: IEEE Computer Society, 46–57.

Rozier KY. 2011. Linear temporal logic symbolic model checking. *Computer Science Review* **5(2)**:163–203 DOI 10.1016/j.cosrev.2010.06.002.

van der Aalst W. 1994. Putting high-level Petri nets to work in industry. *Computers in Industry* **25(1)**:45–54 DOI 10.1016/0166-3615(94)90031-0.

van Dijk T, van de Pol J. 2017. Sylvan: multi-core framework for decision diagrams. *STTT* **19(6)**:675–696 DOI 10.1007/s10009-016-0433-2.

Vardi MY. 1995. An automata-theoretic approach to linear temporal logic. In: Moller F, Birtwistle GM, eds. *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings), volume 1043 of Lecture Notes in Computer Science*. Berlin: Springer, 238–266.

Vardi MY, Wolper P. 1986. An automata-theoretic approach to automatic program verification. In: *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*. Washington, D.C.: IEEE Computer Society, 332–344.

Vardi MY, Wolper P. 1994. Reasoning about infinite computations. *Information and Computation* **115(1)**:1–37 DOI 10.1006/inco.1994.1092.

Visser W, Barringer H. 1998. CTL* Model Checking for SPIN. In: *The 4th International SPIN Workshop*. France: ENST.

Visser W, Barringer H. 2000. Practical CTL* model checking: should SPIN be extended? *International Journal on Software Tools for Technology Transfer* **2(4)**:350–365 DOI 10.1007/s100090050042.

Zurawski R, Zhou M. 1995. Petri net and industrial application: a tutorial. *IEEE Transactions on Industrial Electronics* **41(6)**:567–583 DOI 10.1109/41.334574.