A CTL* model checker for Petri nets

Elvio Gilberto Amparore[1], Susanna Donatelli[1], and Francesco Gallà[1]

[1] Università di Torino, Dipartimento di Informatica, Torino, Italy
(amparore,susi,galla)@di.unito.it

Abstract. This tool paper describes RGMEDD*, a CTL* model checker that computes the set of states (sat-sets) of a Petri net that satisfy a CTL* formula. The tool can be used as a stand-alone program or from the GreatSPN graphical interface. The tool is based on the decision diagram library Meddly, it uses Spot to translate (sub)formulae into Büchi automata and a variation of the Emerson-Lei algorithm to compute the sat-sets. Correctness has been assessed based on the Model Checking Context 2018 results (for LTL and CTL queries), the sat-set computation of GreatSPN (for CTL) and LTSmin (for LTL), and the μ-calculus model checker of LTSmin for proper CTL* formulae (using a translator from CTL* to μ-calculus available in LTSmin). As far as we know, RGMEDD* is the only available Büchi-based CTL* model checker.

1 Introduction

In recent years, the model checking of CTL [10] and LTL [22] temporal logics for (colored) Petri nets(PN) has seen a boost in interest, and several tools and methods have been developed. Efficiency has been a main driving force behind this effort, also motivated by the lively Model Checking Competition (MCC) [19]. MCC models, formulae, and their evaluations are publicly available, making MCC data a very valuable benchmark for (Petri net) tools. The MCC includes LTL and CTL properties, but does not consider CTL* [15] ones. CTL* is a temporal logic strictly more expressive than CTL and LTL. Although various theoretical aspects of CTL* model checking have been extensively studied in the past, very few CTL* model checkers exist, despite the fact that CTL* properties are of practical interests, for example for modelling fairness constraints for CTL properties. It is well known that various forms of fairness constraints are directly expressible in LTL, while this is not the case for CTL.

Algorithms for CTL* model checking can either be based on Büchi automata, as illustrated in [6](page 429), or they can rely on the translation from CTL* into μ-calculus [20], as done in the work of Dam [11, 12], using the standard fixed point iteration of μ-calculus to compute the sat-set.

RGMEDD*, the CTL* model checker described in this paper, computes the sat-sets using a Büchi-based model checking algorithm. Given a CTL* formula, the algorithm identifies the LTL sub-formulae of maximal length and uses Spot [14] to translate each of them into a Büchi automaton. States are encoded as Decision Diagrams (DD), using the Meddly [5] library. RGMEDD* can be run

as a stand-alone tool or as part of the GreatSPN [2] tool suite, called from a "check property" window of the GreatSPN graphical interface [1].

We could only find another Petri net tool that can deal directly with CTL*: LTSmin [18]. This tool translates CTL* into μ-calculus, using the procedures defined in [11, 12]). Note that μ-calculus for Petri nets is avaliable also in TINA [7], but no translator from CTL* to μ-calculus is provided. We could not find any CTL* tool based on Büchi automata. the set of available tools does not change even when looking to model checkers with input languages other than Petri Nets. There are papers on CTL* for Spin [17, 25], but we could not find any implementation available. There is an implementation of μ-calculus for nuXmv [8], which could lead to a CTL* model checker but, for the time being, only CTL* formulae that are either LTL or CTL are actually processed.

Validation of RGMEDD* has been achieved taking advantage of the MCC2018 models, LTL and CTL formulae, and associated truth values for the models' initial state. Computations of the sat-sets of CTL* formulae have been checked in two ways: sat-sets of formulae that are plain CTL have been compared with the sat-set computed by RGMEDD3, the existing CTL model checker of GreatSPN, and LTSmin, while for CTL*-only formulae the comparison has been conducted against LTSmin, using its CTL* module.

We undertook the construction of a CTL* model checker to use it: 1) to test the efficiency of a DD-based implementation of LTL and CTL*; 2) to explore whether a Büchi automata approach can favour the formulation of counterexamples and witnesses; 3) to investigate the efficacy for CTL* of the variable ordering techniques developed in [3]; and 4) to support teaching: following the effort in [4] the users of GreatSPN to experiment (within the same window of the GUI) formulae of multiple logics: LTL, (*fair*)CTL and CTL*.

The paper is organized as follows: Section 2 introduces basic definitions, Section 3 reviews the algorithm for CTL* model checking, the symbolic data structures used to implement it, the tool architecture and the integration into the GreatSPN graphical interface. Section 4 describes the testing performed, while Section 5 concludes the paper.

2 Background

PT-nets. A place-transition (PT) Petri net M is defined [21] as a tuple $M = \langle P, T, A, W, m_0 \rangle$, 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, $W : A \to \mathbb{N}_{\geq 1}$ is the arc weight function, and $m_0 : P \to \mathbb{N}$ 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 of t contain at least $W(p, t)$ tokens. The *firing* of t removes such tokens from the input places, and adds to each output place p' an amount of $W(t, p')$ new tokens. Notation $m \xrightarrow{t} m'$ indicates the firing of t from marking m to m'. The *reachability set* (RS) is the set of all markings reachable from m_0. Figure 1**(A)** shows a simple Petri net model M with 3 places and 4 transitions, with a RS of 4 markings.

GBA. A Generalized Büchi Automaton (GBA) is a tuple $A = \langle Q, \Sigma, \delta, Q_0, \mathcal{F} \rangle$, where Q is a finite set of locations, Σ is a set of atomic proposition labels, $\delta : Q \times \Sigma \to 2^Q$ is a total transition function, $q_0 \subseteq Q$ is the set of initial locations, and $\mathcal{F}$ is a subset of 2^Q and each element of $\mathcal{F}$ is called an acceptance set. Every LTL formula ϕ can be translated into an equivalent GBA [23]. The details of this translation are outside the scope of this paper. Figure 1**(B)** shows a GBA with 3 locations and a single atomic proposition $\#P2{=}1$, corresponding to a boolean formula on M. Location 1 is q_0, and $\mathcal{F} = \big\{\{0\}\big\}$.

CTL*. The language of CTL* formulae of RGMEDD* is inductively defined by:

$$
\begin{aligned}
\Psi &::= \top \mid \bot \mid \text{dead} \mid \text{en}(\tau) \mid \Theta \bowtie \Theta \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \mid \exists\phi \mid \forall\phi \\
\phi &::= \Psi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid X\phi \mid F\phi \mid G\phi \mid \phi U \phi \\
\Theta &::= n \mid \#\text{p} \mid \text{bounds}(\pi) \mid -\Theta \mid \Theta \circ \Theta
\end{aligned}
$$

where $n \in \mathbb{N}$, $p \in P$, $\pi \subseteq P$, $\tau \subseteq T$, $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$ is a comparison operator, and $\circ \in \{+, -, *, /\}$ is an arithmetic operator. The rules Ψ, ϕ and Θ are the rules for the *state*, *path* and *integer* formulae, respectively. The state formula *dead* is a special label for all RS states that do not enable any transition; $en(\tau)$ is satisfied in all RS states enabling at least one transition $t \in \tau$; $\#p$ evaluates to the cardinality of place p in the current marking; $bounds(\pi)$ is the maximum sum of token counts of all places in π in every reachable marking. A CTL* formula with no quantifiers but the initial one is an LTL formula.

Product $RS \otimes A$. Model checking of properties expressed using Büchi automata follows the schema of [24]. A Transition System (TS) is generated from the cross product $RS \otimes A$ between a path-formula GBA A and the RS of M. States of this TS are pairs $\langle m, q \rangle$, with m a Petri net marking and q a GBA location.

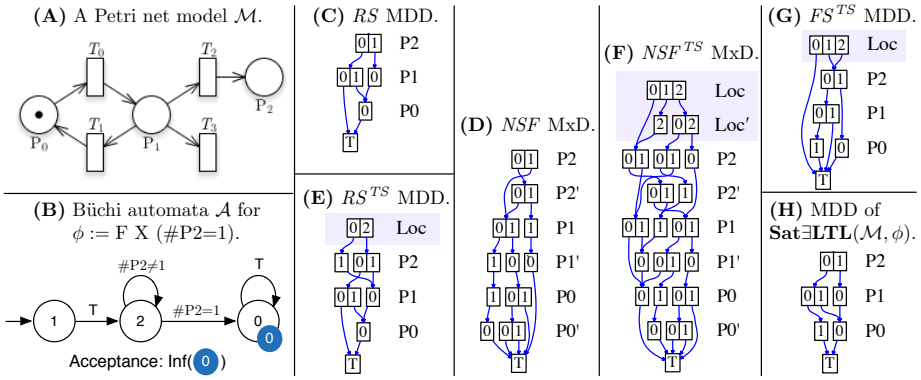

Fig. 1. An example of the MDDs and MxDs generated for CTL* evaluation.

Algorithm 1 Sat-set generation of a maximal proper state subformula ϕ.

procedure SAT∃LTL((M, ϕ))
  $A \leftarrow$ translate ϕ into a Büchi Automaton
  $\langle S_0^{TS}, NSF^{TS}, AS_{\mathcal{F}}^{TS} \rangle \leftarrow$ **BuildTransitionSystem**(M, A)
  $RS^{TS} \leftarrow Saturate(S_0^{TS}, NSF^{TS})$
  switch $type(A)$ **do**
    case StrongBüchi:
      $FS^{TS} \leftarrow RS^{TS} \models E_{fair} G(\mathbf{true}, fair{=}AS_{\mathcal{F}}^{TS})$ ▷ Emerson Lei Algorithm
    case WeakBüchi:
      $FS^{TS} \leftarrow RS^{TS} \models EF\, EG(AS_{\mathcal{F}}^{TS}[0])$
    case TerminalBüchi:
      $FS^{TS} \leftarrow RS^{TS} \models EF(AS_{\mathcal{F}}^{TS}[0])$
  $Sat(\phi) \leftarrow relabel(FS^{TS} \cap S_0^{TS}, none)$
  return $Sat(\phi)$
end procedure

MDD and MxD. Decision Diagrams (DD) are a data structure used to encode large sets of structured data. GreatSPN uses the DD library Meddly [5], which supports, among others, binary (BDD) and multivalued (MDD) DDs. We shall use MDD to encode both the RS of the Petri net, and the TS of the product $RS \otimes A$. MDDs encoding a relation function are called MxD, and are used by the tool to represent transitions and their union as a transition relation. An MxD has twice the levels of an MDD, and in each pair of levels it encodes the before/after relations of a variable (i.e. a place or a location).
Figure 1**(C)** and **(D)** shows the RS and the transition relation of M encoded as a MDD and as a MxD, respectively. A reachable state corresponds to a path in the MDD of **(C)**: so, taking the rightmost path, we have that $(1, 0, 0)$ is a reachable state. The MDD are fully-reduce, so the leftmost path encodes both $(0, 0, 0)$ and $(0, 0, 1)$.

3 The RGMEDD* architecture

Given a Petri net model M and a CTL* formula ψ, the solver first descends recursively through the syntax tree of ψ to extract each *maximal proper state subformula* [6, pp.427], and then model checks each maximal proper state subformula ψ independently, in a bottom up process. Let ψ' be a maximal proper state subformula where all inner maximal proper state subformulae have been replaced with atomic propositions. For example if a and b are atomic propositions, then $\exists \mathrm{XX}(\forall\, a \mathrm{U} b)$ has two maximal subformulas: $\forall\, a \mathrm{U} b$ and $\exists \mathrm{XX} c$, where c is the atomic proposition that stands for the sat-set of $\forall\, a \mathrm{U} b$. If ψ' only contain logical operations, it is trivially checked by applying the corresponding logical functions. Therefore, it remains to treat only the quantified cases $\psi' = \exists\phi$ and $\psi' = \forall\phi$. For the first case the model checker implements a single function to compute the set of states of M satisfying ϕ, denoted Sat∃LTL(M, ϕ). The method Sat∃LTL(M, ϕ)

identifies the set of all markings m that have at least one path starting in m satisfying ϕ, which gives the semantics for the CTL* expression $\exists\phi$. The CTL* expression $\forall\phi$, corresponding to the more usual LTL semantics, is obtained from $\forall\phi = \neg(\exists\neg\phi)$, which is computed as $Sat(\forall\phi) = RS \setminus \text{Sat}\exists\text{LTL}(M, \neg\phi)$.

A pseudo-code of the Sat$\exists$LTL(M, ϕ) function is given in Algorithm 1. The function first translates the path formula ϕ into a GBA A using Spot (line 2). It then encodes (line 3) the transition system $RS \otimes A$: each TS state $\langle m, q\rangle$, is encoded in a MDD with $|P| + 1$ levels. The set of reachable states in $RS \otimes A$ is generated using *saturation* [9] (line 4). An infinite path meets the state-based acceptance condition of a GBA if it visits infinitely often at least one state in each acceptance set $F \in \mathcal{F}$. The work in [16] shows that the set of states originating accepting paths can be computed by the fair CTL formula $E_{fair}G(\mathbf{true}, \mathcal{F})$ on the TS. For some subclasses of GBAs (weak and terminal), a simplified procedure (lines 5–11) can be used [26]. The final set $Sat(\phi)$ is obtained by taking all the fair states (i.e. those that satisfy the acceptance condition of the GBA) that are also initial states of the RS (line 12).

Most of the complexity resides in the generation of $RS \otimes A$ with Decision Diagrams, summarized in Algorithm 2. Its generation requires both the MDD of the RS and the MxD of the transition relation, referred to as the *Next-State-Function* [9] (*NSF*) of the Petri net model. Each edge $q \xrightarrow{s} q'$ of A is encoded as a new MxD (line 5), by modifying the transition relation of the Petri net to reach only $Sat(s)$ markings, and at the same time by moving the GBA location from q to q'. To ensure that the generated transition system is a proper Kripke structure, all deadlock states of the Petri net model must be closed by a self-loop, which is built separately for each edge (lines 6–7). The transition relation NSF^{TS} of the TS is created from the union of all the edge's MxD (line 8). The

Algorithm 2 Encoding of the $RS \otimes A$ transition system.

```
1: procedure BuildTransitionSystem((M, A))
2:     S_0^TS ← MDD()
3:     NSF^TS ← MxD()
4:     for each edge e = q -s-> q' in δ: do
5:         edgeMxD^e ← loc_change(NSF ∩ (RS × Sat(s)), q, q')
6:         b ← Sat(s) ∩ deadlock                                  ▷ Self loops
7:         selfLoopMxD^e ← loc_change(b × b, q, q')
8:         NSF^TS ← NSF^TS ∪ (edgeMxD^e ∪ selfLoopMxD^e)
9:         if q ∈ Q_0 then
10:            S_0^TS ← S_0^TS ∪ relabel(Sat(s), q')
11:        end if
12:    end for
13:    for each accepting set F ∈ F: do
14:        AS_F^S ← ⋃{ edgeForVar(l) | for each l ∈ F }   ▷ MDD of all the locations in F
15:    end for
16:    return ⟨S_0^TS, NSF^TS, AS_F^TS⟩
17: end procedure
```

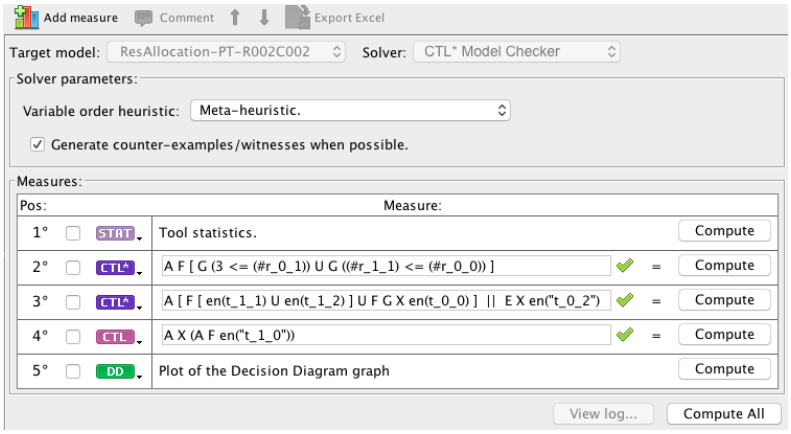

Fig. 2. A screenshot of the CTL/CTL* interface of RGMEDD* in GreatSPN.

function $relabel(d, q)$ takes a MDD d and sets the location level to q, while the $loc\_change(x, q, q')$ takes a MxD x and replaces for the location level the relation $q \to q'$. The encoded TS is a tuple made by the transition relation NSF^{TS}, the set of initial states S_0^{TS} of the TS, which is the set of markings that are accepted by an edge leaving an initial location of the GBA (line 10), and the encoding $AS_{\mathcal{F}}^{TS}$ of the accepting sets $\mathcal{F}$ of the TS as MDDs (line 14).

Example. Figure 1 shows in **(E)** the MDD of RS^{TS}, generated as a fixed-point image of the MxD NSF^{TS} shown in **(F)**. The DDs of $RS \otimes A$ encode both the places of M and the locations of A, and therefore have an additional level of nodes. The set of fair states FS^{TS} visited infinitely often are also encoded as a MDD, shown in **(G)**. Finally **(H)** shows the MDD of the sat-set of the CTL* formula $\exists\, \phi$, encoding the 3 satisfying markings (all markings of RS except the one where all places have zero tokens).

Tool. The CTL* model checker is available both as a command line tool and inside the integrated GUI. Figure 2 shows the interface. For each model, a list of queries can be inserted, following the grammar given in Section 2. Queries are declared as either CTL or CTL*, which changes the model checking algorithm used. The algorithm described in Section 3 is used for CTL* queries only.

4 Testing

We have tested different aspects of RGMEDD*: the *correctness* of the results and the *performance* of the tool. MCC2018 models and relative model instances and formulae were used. We have considered only instances for which GreatSPN is able to build the RS, since the RS is required for sat-set computation.

The reported tests are the *final* ones, but the MCC instances have been extensively used also during the debugging phase allowing to discover both technical errors (in the implementation of the algorithm) and semantics ones (different behaviour for systems with deadlocks). Indeed Petri net models with deadlocks do not feature only infinite paths, as per the semantics of LTL and CTL. A standard way to turn around this problem is to "stutter" deadlock states by adding a self-loop. MCC assumes that deadlock states are stuttered only for LTL model checking, since this is required by the Büchi-based construction. RGMEDD* assumes deadlocks are stuttered, which may cause discrepancies when comparing RGMEDD* on CTL formulae. Therefore the MCC instances considered have been split in two sets: deadlock-free models (DFM), with 408 instances, and non deadlock-free models (DM), with 539 instances.

4.1 Testing of truth values for CTL and LTL formulae

This test is based on the MCC2018 results for four categories of queries: *CTLCardinality*, *CTLFireability*, *LTLCardinality* and *LTLFireability*. Each category has 16 queries. A pair $\langle \textit{model instance}, \textit{category} \rangle$ is called an *examination*. For each examination an *expected result* is provided, that was used to check the correctness of RGMEDD*. To limit resource consumption we set a time limit of 300s and a memory limit of 2GB. With these limitations we could complete 360 examinations, that is to say 5760 ($=360 * 16$) queries. For LTL categories we have used instances of both DFM and DF models, while for CTL only DFM ones have been used. Over all the 5760 tests performed we have got a single mismatch, but this is query for which there is a mismatch also among the three tools that were able to evaluate the query in MCC2018.

4.2 Testing of sat-sets computation of CTL formulae

GreatSPN has a CTL model checker called RGMEDD3 that recursively computes the sat-set of CTL formulae. With a timeout of 60 seconds and 2GB of memory, RGMEDD3 completes 3544 queries from 168 different model instances from the DF set. RGMEDD* timeout-out on 10.05% of the queries. RGMEDD3 was faster than RGMEDD* in 92.65% of all queries completed by both tools.

Fig. 3 (A) shows the execution times of the two tools, each query being a dot. In all completed tests the tools produce sat-sets of equal cardinality.

4.3 Assessment on CTL* formulae.

Here we report the assessment of RGMEDD* against LTSmin, run using the *pins2lts-sym* interface with the *–ctlstar* option which first converts CTL* into μ-calculus (which may incur an exponential cost). Also LTSmin is based on decision diagrams (of the Sylvan [13] library) and to make a more realistic comparison we have enforced the use of the same variable order by the two tools. Before checking CTL* formulae we have tested their behaviours on known results.

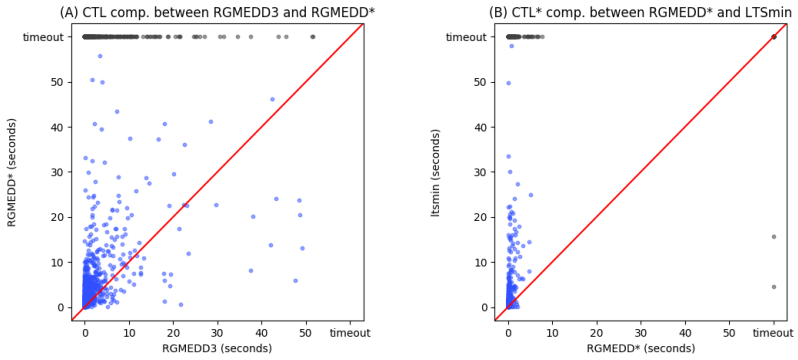

Fig. 3. Execution times of RGMEDD* vs. RGMEDD3 (A) and vs. LTSmin (B).

1. Tool validation against MCC results. We computed the sat-sets generated by RGMEDD* and pnml2lts-sym when provided with the same queries from the MCC examination of LTLCardinality and CTLCardinality (DF only models) set for which the RS can be built. With the same resource limits as before, of the 1248 formulae considered (covering instances from 32 different models), RGMEDD* completed 1162 and LTSmin 702. Among these 702 formulae successfully completed by both model checkers, there were 7, from different models, over which the two tools did not agree: For these formulae MCC states that they are true in the initial marking, RGMEDD* returns sat-sets that include the initial marking, while LTSmin returns empty sat-sets, which seems a wrong answer.

2. Relative performances of the two tools. We checked the time performance of the two tools on the same instances as above, excluding the time for RS computation. Fig. 3 (B) shows the comparison. RGMEDD* performs better in average and a closer look at LTSmin times reveals that a significant amount of time is spent converting CTL* formulae to μ-calculus.

3. Correctness and performance on CTL formulae.* Since there is no available set of CTL* formulae for the MCC models (and for any other models we could find) we have generated formulae algorithmically by parsing *CTLCardinality* queries from MCC2018 and deleting each path quantifiers with a probability of 70%. The first quantifier is always kept, in order to preserve consistency with the CTL* grammar. For these tests, which are mainly aimed at checking correctness, we considered only 39 models whose RS are rather quick to generate. This gives a total of 624 queries, and on 218 both tools complete with the given resources. RGMEDD* timed out in 5.44% of the queries while LTSmin timed out in 65.06% of the queries. RGMEDD* was faster than LTSmin in 85.78% of all queries completed by both tools. More importantly the cardinality of the sat-sets coincides for both tools.

5 Conclusion

RGMEDD* is the new CTL* model checker of GreatSPN. It leverages two libraries: Spot for the translation from CTL* to Büchi automata and Meddly for decision diagram manipulation. RGMEDD* allows GreatSPN users to compute the set of reachable states that satify CTL* and LTL properties. The approach is fully integrated into the GreatSPN GUI, that already included the possibility of computing the sat-set of CTL formulae. RGMEDD* itself can be used to check also CTL properties, although our testing has confirmed that, as expected, the standard CTL model checking algorithm has superior performances.

Testing of RGMEDD* had to face a number of difficulties, due to the availability of a single CTL* tool, LTSmin. To be able to use LTSmin for our benchmark it was first necessary to fix a few syntactic problems in the LTSmin parsing of CTL* formulae: nevertheless it was an easy to use tool and we observed very limited discrepancies in the results. Although both tools are based on decision diagrams, RGMEDD* seems to perform significantly better than LTSmin, that suffers for the expensive translation from CTL* into μ-calculus.

Based on the experience gained in building RGMEDD* we plan to develop a model checker for the fair variant of CTL (reusing the available implementation of the Emerson-Lei algorithm for $E_{fair}G\phi$). We also plan to work on the generation of counterexamples and witnesses: these two topics are of paramount importance for the verification of distributed algorithms.

Finally, we shall work on improving memory and time performance by avoiding, as much as possible, the construction of a single monolithic decision diagram for the next state function of the Petri Net and of the $RS \otimes A$ transition system.

Availability. A virtual machine with the tool pre-installed can be downloaded from `http://www.di.unito.it/~greatspn/VBox/RGMEDDstar-vm.ova`. The source code of GreatSPN is available from `https://github.com/greatspn/SOURCES`.

Acknowledgements. We would like to thank Jaco van de Pol for the various insights given on LTSmin, and Yann Thierry Mieg for the discussion on finite paths and model checking.

References

1. Amparore, E.G.: A new GreatSPN GUI for GSPN editing and CSL^{TA} model checking. In: Proc of the 11th QEST Conf. vol. 8657 LNCS, pp. 170–173. Springer (2014)
2. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 Years of GreatSPN, chap. In: Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi, pp. 227–254. Springer (2016)
3. Amparore, E.G., Ciardo, G., Donatelli, S., Miner, A.S.: *i-Rank*: A variable order metric for DEDS subject to linear invariants. In: 25th Int. Conf., TACAS 2019. LNCS, vol. 11428, pp. 285–302. Springer (2019)
4. Amparore, E.G., Donatelli, S.: GreatTeach: A Tool for Teaching (Stochastic) Petri Nets. In: 39th ATPN Int. Conf. LNCS, vol. 10877, pp. 416–425. Springer (2018)
5. Babar, J., Miner, A.: Meddly: Multi-terminal and Edge-Valued Decision Diagram LibrarY. In: Proceedings of QEST Conf. pp. 195–196. IEEE (2010)

6. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
7. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. Int. Journal of Production Research 42, 2741–2756 (07 2004)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: CAV Int. Conf. LNCS, vol. 8559, pp. 334–342. Springer (2014)
9. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: TACAS. pp. 328–342 (2001)
10. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer (1981)
11. Dam, M.: Translating CTL* into the modal μ-calculus. Tech. Rep. ECS-LFCS-90-123, University of Edinburgh (1990)
12. Dam, M.: CTL* and ECTL* as fragments of the modal μ-calculus. Theoretical Computer Science 126(1), 77 – 96 (1994)
13. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. STTT 19(6), 675–696 (2017)
14. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Proceedings of the 14th Int. Symp. of ATVA. LNCS, vol. 9938, pp. 122–129. Springer (2016)
15. Emerson, E.A., Halpern, J.: "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. JACM 33(1), 151–178 (1986)
16. Emerson, E.A., Lei, C.: Efficient model checking in fragments of the propositional mu-calculus. In: Logic in Computer Science (LICS'86). pp. 267–278. IEEE (1986)
17. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (01 2004)
18. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015)
19. Kordon, F., et al.: Presentation of the 9th Edition of the Model Checking Contest. In: Proc. of TACAS. LNCS, vol. 11429, pp. 50–68. Springer (2019)
20. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27(3), 333 – 354 (1983)
21. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
22. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57. IEEE Computer Society (1977)
23. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G.M. (eds.) Logics for concurrency, LNCS. vol. 1043, pp. 238–266. Springer (1995)
24. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and computation 115(1), 1–37 (1994)
25. Visser, W., Barringer, H.: CTL* Model Checking for Spin. In: The 4th International Spin Workshop. ENST, France (November 1998)
26. Wang, C., Hachtel, G.D., Somenzi, F.: Abstraction refinement for large scale model checking. Springer Science & Business Media (2006)