EasyChair Preprint

№ 11276

EnPAC: Petri Net Model Checking for Linear Temporal Logic

Zhijun Ding, Cong He and Shuo Li

November 10, 2023

EnPAC: Petri Net Model Checking for Linear Temporal Logic

Zhijun Ding
Department of Computer Science and Technology
Tongji University
Shanghai, China
dingzj@tongji.edu.cn

Cong He
Department of Computer Science and Technology
Tongji University
Shanghai, China
1105585684@qq.com

Shuo Li*
Department of Computer Science and Technology
Tongji University
Shanghai, China
lishuo20062005@126.com

***Abstract*—In linear temporal logic (LTL) model checking using Petri nets, two important aspects are state generation and exploration for counterexample search. Traditional state generation involves updating a structure of enabled transitions and frequently encoding/decoding to read each encoded state, which can be expensive. This paper proposes an optimized approach that calculates enabled transitions on demand using a dynamic fireset, avoiding the need for such a structure. Additionally, a set of direct read/write (DRW) operations on encoded markings is proposed to speed up state generation and reduce memory usage without the need for decoding and re-encoding. Heuristic information is incorporated into the Büchi automaton for counterexample search to guide exploration toward accepted states. These strategies improve existing methods for LTL model checking with Petri nets. The optimization strategies are implemented in a tool called EnPAC (Enhanced Petri-net Analyser and Checker) for linear temporal logic and evaluated on MCC (Model Checking Contest) benchmarks, demonstrating a significant improvement over previous methods.**

***Index Terms*—Petri nets, Model Checking, State explosion, Encode, Heuristic, Linear Temporal Logic (LTL)**

I. Introduction

Model checking is a highly automatic technology based on a formalism, like Petri nets [1], for verifying finite-state concurrent systems [2]. Actually, many important temporal characteristics or functional requirements of concurrent systems are specified by linear temporal logics (LTLs) [3]. In traditional LTL model checking, a formal system model is synchronized by using the product construction with Büchi automaton [3] representing all behaviors that violate an LTL formula. Then, the existence of a run with infinitely many occurrences of an accepting state in the product automaton provides a counterexample for the LTL formula [4] with an on-the-fly framework [5].

However, the state-explosion problem [6] is the main obstacle to practical model checking, as the number of reachable states is exponentially larger than the size of a system description via Petri nets. Even the complexity of LTL model checking is exponential. So far, a lot of reduction techniques can decrease the size of the state space. Also, many mature tools (e.g., LoLA [7]) of Petri net model checking implement efficient state generation and exploration techniques. This paper focuses on optimizing state generation and exploration strategies in existing Petri nets model checking tools.

Since model checking is essentially an exhaustive exploration technology on state space, state generation is the core of the whole process. Traditional methods must calculate and store all enabled transitions under each reachable state. However, many enabled transitions may never occur under on-the-fly exploration. Thus, computing all transitions and storing all enabled transitions leads to a waste of time and memory. Addressing this problem, LoLA designs a data structure [7] to accelerate computing all enabled transitions $T(m)$ by briefly updating $T(m')$ when migrating from a state m' to the next state m. Although it can avoid computing all transitions by such a static structure, it brings some memory cost. This paper optimizes the calculation of enabled transitions under each state. It is more efficient to calculate the enabled transitions on demand. This paper proposes the first optimized strategy of dynamically calculating the transition set, where only one enabled transition is calculated when a successor state is generated.

Most explicit-state Petri net model checking tools exploit various encoding strategies in marking storage, saving large memory costs. Then, when calculating the enabled transitions, they require reading the number of tokens of particular places in the encoded marking. To our knowledge, they should have a decoding and encoding procedure when reading or writing encoded markings as shown in, e.g., LoLA [8]. However, frequent decoding and re-encoding based on an encoding strategy can reduce tool efficiency. It is challenging to read the number of tokens directly by reading and writing encoded markings. This paper defines a reading and writing pattern for each place and proposes a set of bitwise operations on our new pattern and the encoded markings. Thus, this paper proposes the second optimized strategy of direct read/write (DRW) operations.

State exploration (counterexample search) is another core for explicit LTL model checking. Actually, the faster the counterexample is found, the fewer states are generated with an on-the-fly framework. Thus, it is better to reach an acceptable

This work is supported by National Key Research and Development Program of China under Grant No.2022YFB4501700.

* Corresponding author

state faster than random exploration. Based on this insight, this paper presents the third optimized strategy to add a heuristic to the Büchi automaton. The heuristic can guide the on-the-fly exploration in the direction of accepted states.

Based on the above optimization insights, we implement an explicit-state model checking tool EnPAC, standing for Enhanced Petri-net Analyser and Checker. It can be used for large concurrent systems, modeled as Petri nets [9] or Colored Petri Nets (CPNs) as its colored extension [10]. It can evaluate arbitrary queries specified in linear temporal logic. Then, we evaluate the performance on the benchmarks in Model Checking Contest (MCC) [11]. These optimized state generation and exploration strategies help EnPAC make excellent progress and drastic improvement on the benchmarks of MCC [11].

The contributions are summarized as follows.

1. This paper proposes a dynamic fireset to calculate the enabled transitions on demand. Not only can it avoid traditional complete enabled transition calculation, but it also doesn't require additional data structures like LoLA, one of the state-of-the-art Petri net model checking tools.

2. This paper proposes a new reading and writing pattern for each place on each encoded marking by a set of bitwise operations, called DRW operations, without frequent decoding and encoding on the encoded marking storage.

3. A heuristic Büchi automaton is presented to guide the exploration for searching a counterexample faster, which helps avoid traditional random exploration.

The preliminaries are introduced in Section II and detail the proposed optimizations in Section III. Our experiment results are evaluated based on EnPAC in Section IV. Finally, this paper is concluded in Section V.

II. Preliminary

A. Petri Nets

Petri nets have been widely used in modeling and verifying concurrent systems for many interesting properties, such as deadlock, liveness, and reachability. The definition of Petri net is introduced.

Definition 1: A Petri net N is a five-tuple $N = \{P, T, F, W, m_0\}$ where P is a finite set of places, T is a finite set of transitions (disjoint to P), $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of arcs, $W\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is a weight function where $(x, y) \notin F \iff W(x, y) = 0$, and m_0 is the initial marking. A marking is a mapping $m : P \to \mathbb{N}$.

Definition 2: A transition t is enabled under a marking m if $\forall p \in P, W(p, t) \leq m(p)$. The set of all enabled transitions in m is called *fireset*, denoted by $T(m)$. Firing an enabled transition t under a marking m leads to a new marking m' where $m'(p) = m(p) - W(p, t) + W(t, p)$. This firing relation is denoted as $m \xrightarrow{t} m'$. If there exists a transition sequence $\omega = t_1 t_2 \cdots t_n$ such that $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} \cdots \xrightarrow{t_n} m_n$, m_n is reachable from m_1, written $m_1 \xrightarrow{*} m_n$. The state space of a Petri net consists of $R(m_0) = \{m \mid m_0 \xrightarrow{*} m\}$.

B. Linear Temporal Logic

We define the syntax and semantics of atomic proposition based on MCC [11], and then LTL.

Definition 3: Let $\langle atomic \rangle$ be an atomic proposition, and $\langle int\text{-}expression \rangle$ be an expression evaluated by an integer.

$$\begin{aligned} \langle atomic \rangle &:= is\text{-}fireable(t_1, \cdots, t_n) \\ &\quad |\langle int\text{-}expression \rangle \\ &\quad \leq \langle int\text{-}expression \rangle \\ \langle int\text{-}expression \rangle &:= Int | tokens\text{-}count(p_1, \cdots, p_n) \end{aligned}$$

$is\text{-}fireable\,(t_1, \cdots, t_n)$ holds if either t_1 or t_2 or $\cdots$ or t_n are enabled, and $tokens\text{-}count(p_1, \cdots, p_n)$ returns the exact number of tokens contained in the place set $\{p_1, \cdots, p_n\}$.

Definition 4: Every atomic proposition is an LTL formula. If φ and ψ are LTL formulae, so are $\neg\varphi$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $X\varphi$, $F\varphi$, $G\varphi$, $(\varphi U \psi)$, $(\varphi R \psi)$. Let AP be a non-empty finite set of atomic propositions, $\xi = x_0 x_1 x_2 \cdots$ be a sequence over alphabet 2^{AP}, φ and ψ be LTL formulae. ξ_i is the suffix of ξ starting at x_i. ξ satisfies an LTL formula according to the following inductive scheme: $\xi \models p \iff p \in x_0,\ p \in AP$; $\xi \models \neg\varphi \iff \xi$ dissatisfy φ; $\xi \models \varphi \vee \psi \iff \xi \models \varphi$ or $\models \psi$; $\xi \models X\varphi \iff \xi_1 \models \varphi$; $\xi \models \varphi U \psi \iff i \geq 0,\ \xi_i \models \psi \wedge (\forall j < i,\ \xi_i \models \varphi)$; Other operators $(\wedge, R, F, G)$ can be derived from the above operators: $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$; $\varphi R \psi \equiv \neg(\neg\varphi U \neg\psi)$; $F\varphi \equiv (TRUE)\, U \varphi$; $G\varphi \equiv \neg(F\neg\varphi\ )$

C. On-the-fly Exploration of LTL Model Checking

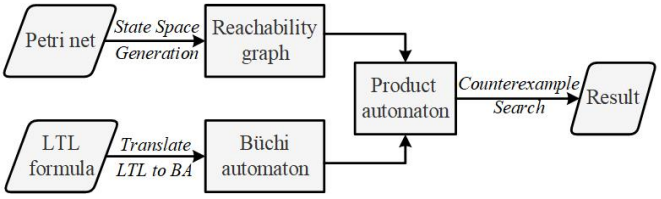


Fig. 1: The entire process of explicit-state LTL model checking

As shown in Fig. 1, explicit-state LTL model-check tools for Petri nets calculate the reachability graph of a Petri net, transform the negative LTL formula into the Büchi automaton, and then generate the product automaton in the form of Cartesian product of reachability graph and Büchi automaton, and finally searches counterexamples on the product automaton. If a counterexample is explored, $false$ is returned. Otherwise, $true$ is returned. More details are in [12].

This process usually uses an on-the-fly framework to optimize the above process. On-the-fly exploration [5] consists of constructing a reachability graph and product automaton while checking for the counterexamples in the product automaton. An advantage of on-the-fly exploration is that it can return a result before the entire state space is constructed.

III. Optimization Strategies

Since explicit model checking is a state exploration technology, the efficiency of state generation and exploration (counterexample search) directly affects its performance. Two optimizations can make state generation much faster with less memory consumption. One is dynamic calculating enabled transitions. The other is direct reading and writing encoded markings without frequent decoding and re-encoding procedures. Concerning state exploration, a heuristic Büchi automaton as an optimization can guide on-the-fly exploration to explore counterexamples in the direction of accepted states.

A. Dynamic Fireset

Enabled transitions play a fundamental role in the state generation since they determine all successor states of a reachable state. The common practice is generating all enabled transitions simultaneously and firing one by one to enumerate all possible successor states. Only one enabled transition is generated at a time in a state. Then, when on-the-fly exploration backtracks to an explored state, how to directly generate the next enabled transition without repeated exploration is a difficulty.

To solve this difficulty, a total order $(\prec, T)$ on the transition set is defined to check which transition is enabled in turn in that order. Also, an array is defined to store all the transitions and use their index as their total order. In addition, each explored state needs to record the last fired transition. In this case, when on-the-fly exploration backtracks to an explored state, it can check the transition just next to the last fired transition by order $(\prec, T)$ to fire the next enabled transition. Once an enabled transition t is found, on-the-fly exploration stops to generate a successor state m by firing t and continues a depth-first search on m. Our new method is named *dynamic fireset* (abbreviated as DYN). Its advantage is that it saves memory and time to calculate and store all enabled transitions under each reachable state.

B. DRW Operation on Encoded Markings

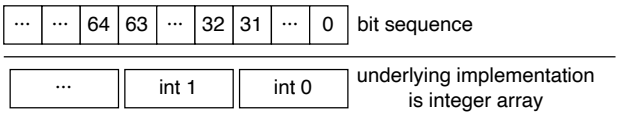


Fig. 2: Underlying implementation of bit sequence

Our encoding strategy is designed on an integer array, as shown in Fig. 2, which is an underlying implementation of a bit sequence. Based on it, this paper proposes a new method of DRW operation on the encoded marking, which can be divided into two sub-tasks. One is to locate, i.e., in which integer the place's coding resides and from which bit of that integer it begins. The other is to read/write its value. Each place's start position is recorded in the bit sequence and length to locate the correct position. A reading and writing pattern for each place is defined to read or write a correct value. Then, the token

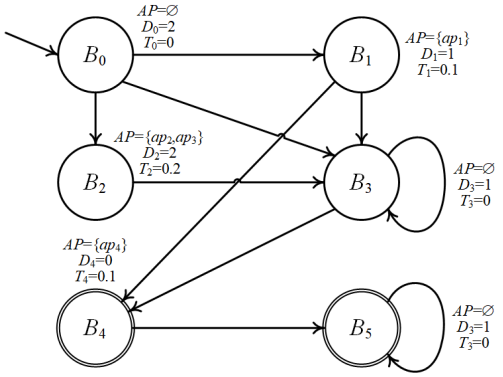


Fig. 3: Büchi automaton with heuristic information

counts can be easily read or written by a series of bitwise operations using these patterns.

There are four kinds of marking encoding in APPENDICES A-A of [13]. NUPN [14] encoding is used to illustrate our DRW operations. Each place carries two extra attributes, $myunit$, and $myoffset$, indicating the unit in which it is located and the offset number in the unit, respectively. Each unit carries two attributes, too, $startpos$ and $unitlen$, indicating the start position in the bit sequence and how many bits this unit takes. When reading or writing a place, there are two cases. In other words, the encoding of the unit occupies only one integer or spans two integers. The algorithms for the two cases are detailed in APPENDICES A of [13].

C. Heuristic Büchi Automaton

Before state exploration, the Büchi automaton is automatically generated, which has complete information. This paper proposes the heuristic in Büchi automaton in two aspects. Firstly, searching counterexamples in a product automaton is to find a strongly connected component containing accepted states. Whether a product state (m_i, b_i) is accepted is determined by its Büchi state part b_i. When generating a reachable state and seeking a Büchi state to combine, it should choose the state that reaches an accepting state the fastest. Thus, the distance to an accepting state is the first aspect. As for each state in the Büchi automaton, the number of atomic propositions affects how easy to synthesize this state.

Based on these insights, this paper adds two extra attributes D_i and T_i, as the heuristics in each Büchi state. Concretely, D_i is the length of the shortest path from state B_i to an acceptable state. Take Fig. 3 as an example, D_0 in B_0 is 2 because its shortest path to an accepted state is $B_0 \rightarrow B_1 \rightarrow B_4$ (or $B_0 \rightarrow B_3 \rightarrow B_4$) whose length is 2. And D_i can be computed by Dijkstra's algorithm. Let AP_i be the set of atomic propositions carried by Büchi state B_i, and T_i $|AP_i|*0.1$ (T_i is the number of atomic propositions carried by B_i. The coefficient '0.1' is from our experience on MCC Benchmark). T_i indicates how tough it is for state B_i to produce a reachable state into a product state.

In our heuristic Büchi automaton (abbreviated as HBA), when choosing a Büchi state for the product with a reachable

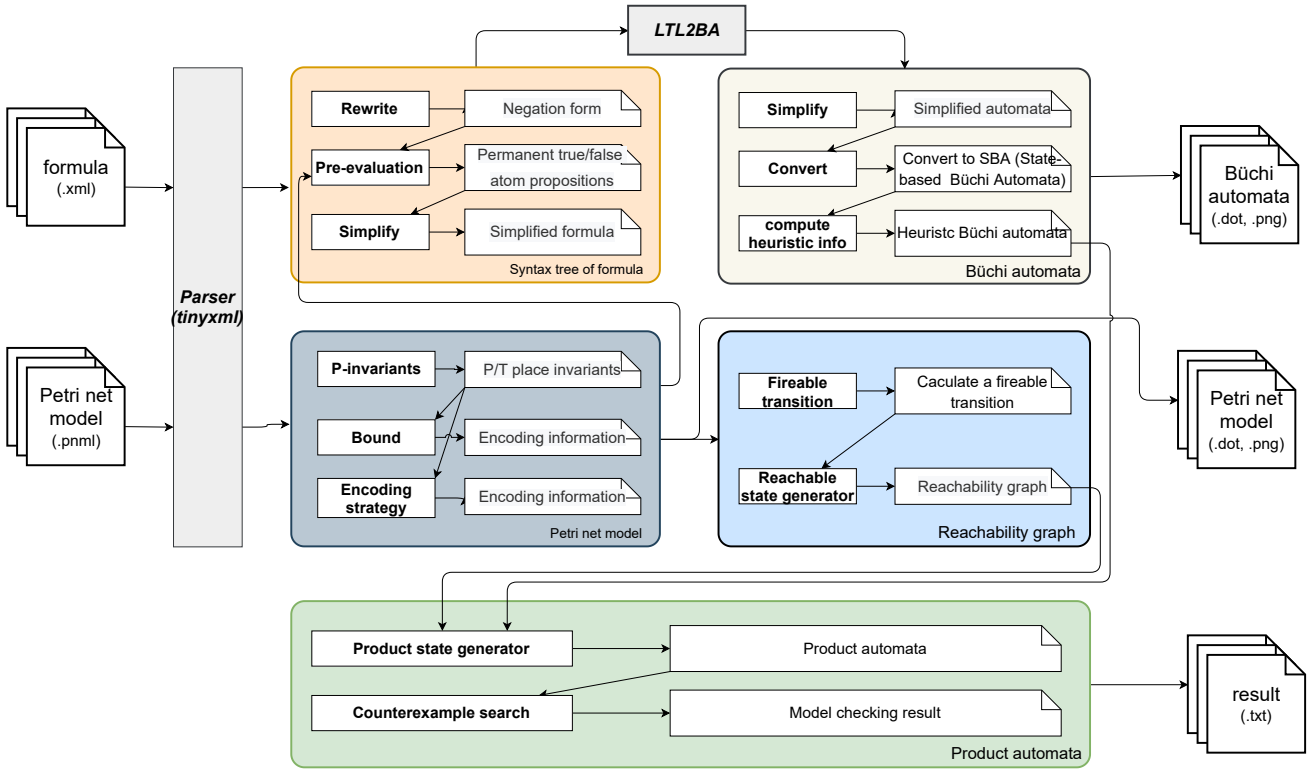


Fig. 4: The architecture of EnPAC

state, this paper prioritizes the state with smaller $D_i + T_i$. It means we always prefer the path that can reach an accepted state fast and be smooth enough.

IV. Experimental Evaluation

A. Installation and Usage

We implement our optimizations in EnPAC (Enhanced Petri-net Analyser and Checker). It is divided into five modules, including the Petri net model, Reachability graph, Syntax tree of LTL formula, Büchi automaton, and Product automaton. The architecture of EnPAC is shown in Fig. 4, detailed in APPENDICES A of [13].

EnPAC can be downloaded from https://github.com/Tj-Cong/EnPAC_2021 and installed easily. The GitHub homepage presents a user manual that describes the installation procedure, file formats, output, and options. EnPAC can be utilized on the command line of the Linux terminal. The results can be displayed on the screen or in a file.

B. Benchmarks and Methodology

For evaluating the optimization strategies via EnPAC, we use the benchmarks provided by MCC [11]. The benchmarks consist of 1016 Petri net instances, as well as 32 LTL formulae per instance (32512 LTL formulae).

In the benchmarks, there are 3672 LTL formulae that no tool could give a result in MCC'2020 [15]. EnPAC has not yet implemented *dynamic fireset* (abbreviated as DYN), *direct read/write operation* (abbreviated as DRW), and *heuristic Büchi automaton* (abbreviated as HBA) for MCC'2020 [15]. Thus, we compare our three optimizations with the results in MCC'2020 [15] to ensure that the results are persuasive. We call the version without the implementation of our optimizations the *original* method (abbreviated as ORI).

C. Experimental Analysis

For verifying each formula, there is a time limit of 300 seconds and a memory limit of $16GB$. We use each optimization individually to illustrate their performances. The experimental results are shown in APPENDICES B of [13].

1) Experiments for Dynamic Fireset: In order to show the effect of DYN clearly, each formula's time and memory peak is recorded. TABLE III of [13] shows the comparison results between the dynamic fireset method (DYN) and the original method (ORI) on 8 Petri nets instances (their names are in the first column). Two LTL formulae are verified for each instance in the second column. Concretely, T_{ORI} and T_{DYN} are the whole time of ORI and DYN, respectively. And M_{ORI} and M_{DYN} are the memory peak of ORI and DYN, respectively. To quantify the optimized performance for DYN, we calculate ∇T_1 by T_{ORI}/T_{DYN}, and ∇M_1 by M_{ORI}/M_{DYN} in TABLE III of [13]. The average of each result is shown in the last row. All results come to the same conclusion that our DYN outperforms ORI on time and memory consumption.

It can be found from the experimental results that our DYN method is slightly faster than the ORI method in most instances. In particular, an LTL formula of 'CircadianClock-PT-001000' is originally timed out, but our optimization of

DYN can output the result within 130s. Except for this result of a timeout, DYN has an average improvement on time of 3.41 times. Moreover, because DYN does not need to store all enabled transitions in every reachable state, it uses much less memory consumption than ORI.

2) Experiments for DRW Operations: Due to different DRW operations for our encoding strategies as explained in APPENDICES A-A of [13], we conduct separate experiments on 1-safe encoding (8 instances), NUPN encoding (10 instances), and P-invariant encoding (5 instances) in the first column. For each instance in the second column, two LTL formulae are also in the third column. T_{ORI} and T_{DRW} are the whole time of ORI and DRW, respectively. And M_{ORI} and M_{DRW} are the memory peak of ORI and DRW, respectively. To quantify the optimized performance for DRW, we calculate ∇T_2 by T_{ORI}/T_{DRW} and ∇M_2 by M_{ORI}/M_{DRW} in TABLE IV of [13]. The average of each result is shown in the last row.

It can be found that DRW is much faster than ORI on time. For 1-safe encoding, DRW outperforms ORI by more than 20 times in 9 formulae. Especially for NUPN encoding, DRW outperforms ORI in all formulae on time. In P-invariant encoding, DRW outperforms ORI by more than 200 times in 4 formulae. And DRW has an average improvement of 245.52 times than ORI. However, our DRW method uses slightly more memory because it requires additional space overhead for the read/write patterns of each place. But such costs are minuscule since the average of ∇M_2 is mostly close to 1.

3) Experiments for Heuristic Büchi Automaton: In addition to time and memory, we add a comparison of the reachable states that need to be generated to find counterexamples. It can reflect whether the heuristic Büchi automaton can guide on-the-fly exploration to find the counterexample faster. TABLE V of [13] shows the experimental results on 10 instances in the first column with two verified LTL formulae in the second column. Concretely, N_{ORI} and N_{HBA} are the state counts, T_{ORI} and T_{HBA} are the whole time, and M_{ORI} and M_{HBA} are the memory peak of ORI and HBA, respectively. We also calculate ∇N by N_{ORI}/T_{HBA}, ∇T_3 by T_{ORI}/T_{HBA}, and ∇M_3 by M_{ORI}/M_{HBA}.

In TABLE V of [13], there are 4 formulae that are originally timed out. And HBA outputs the results successfully with few states. Obviously, our heuristic Büchi automaton helps find counterexamples faster. Due to generating fewer states, they also consume less memory. The average of ∇T_3 is 6.4. Although the heuristic information does not lead well to finding the counterexample for many other formulae, it does not produce excessive time and memory costs. Most of ∇M_3 are 1.00.

4) Discussion: We sketch four scatter plots in Fig. 5 on the benchmarks of MCC [11]. The x-axis denotes the time/memory of DRW, DYN, and HBA, while the y-axis denotes the time/memory of ORI. In the scatter plot, each dot represents an LTL formula verification, and the dots above the diagonal lines are the winning cases of our optimization.

In Fig. 5(a) and (b), the time and memory of DRW are demonstrated on all encoding methods, where 'P-invariant' represents P-invariant encoding, 'NUPN' represents NUPN encoding, and '1-safe' represents 1-safe encoding. We can see that DRW can significantly reduce time on most LTL formulae. And the scores for DRW are the formula counts that ORI cannot output the result within $300s$ (although they have less memory in Fig. 5(b)). It can be found that DRW can output the results within $150s$.

In Fig. 5(c) and (d), the scores for DYN and HBA are also the formula counts that ORI cannot output the result within $300s$. Obviously, HBA works extremely well on some formulae, as we can see that some orange triangles and circles are much higher above the diagonal lines. Most orange dots are distributed near the diagonal lines. It confirms HBA does not produce large excessive costs in time and memory. HBA can also be counterproductive on individual formulae because it does not lead to counterexamples. Although most blue dots are distributed near the diagonal lines in Fig. 5(c), DYN is much more effective in optimizing memory based on Fig. 5(d).

There are $3,672$ formulae that no tool can output the results in MCC'2020 [15]. After using three optimization strategies, EnPAC can give results for 432 unknown formulae. Under the complete benchmarks in MCC'2020 [15], there are 28781 LTL formulae that EnPAC has given the correct result before. With three optimizations, EnPAC correctly gives the results for 31735 LTL formulae on the same benchmarks. Thus, our optimization strategies can improve EnPAC by nearly $3,000$ scores, which shows a drastic improvement in EnPAC.

V. Conclusion

A dynamic fireset (DYN) is proposed to save the storage and time of computing some redundant enabled transitions. Direct read/write (DRW) operations on encoded markings are proposed to save the large overhead of decoding and re-coding. In terms of state exploration, the heuristic information to the Büchi automaton (HBA) can guide the search of counterexamples, which speeds up the exploration. We implement a Petri nets tool called EnPAC for verifying LTL and then evaluate it on the benchmarks of MCC.

In the future, we improve the performance of EnPAC with parallel algorithms and more heuristics on the reachability graph or Büchi automaton.

References

[1] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

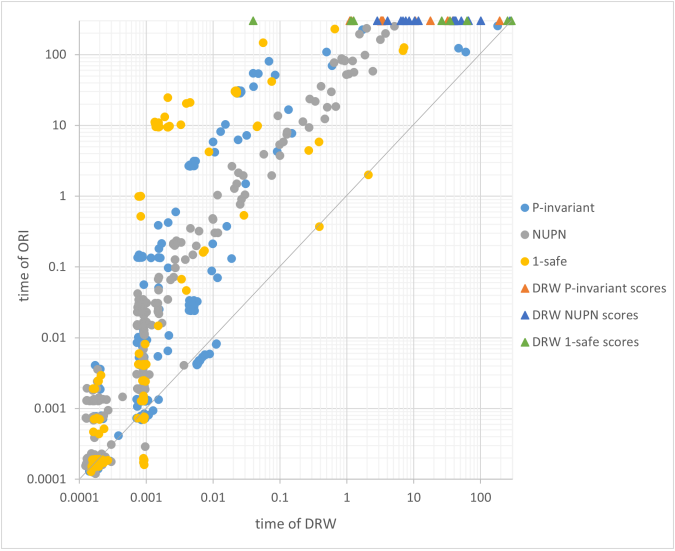
[2] K. Wolf, *How Petri Net Theory Serves Petri Net Model Checking: A Survey*. Transactions on Petri Nets and Other Models of Concurrency XIV, 2019.

[3] P. Gastin and D. Oddoux, "Fast ltl to büchi automata translation," in *International Conference on Computer Aided Verification*. Springer, 2001, pp. 53–65.

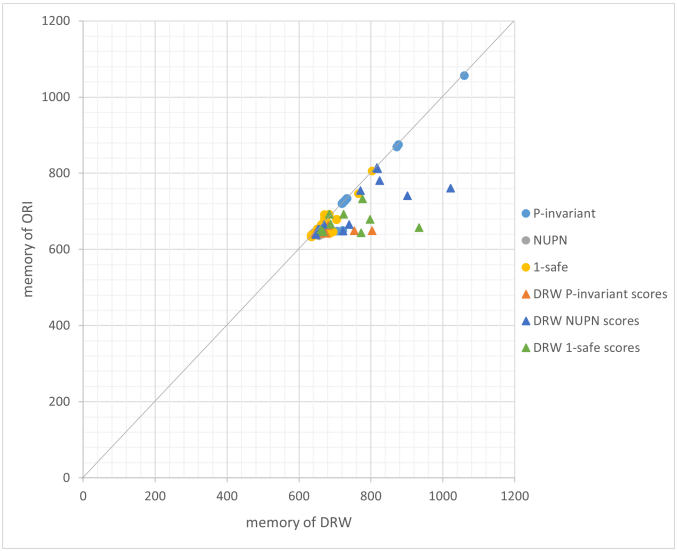
[4] M. Y. Vardi, *An automata-theoretic approach to linear temporal logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 238–266. [Online]. Available: https://doi.org/10.1007/3-540-60915-6_6

[5] J. Geldenhuys and A. Valmari, "More efficient on-the-fly ltl verification with tarjan's algorithm," *Theoretical Computer Science*, vol. 345, no. 1, pp. 60–82, 2005.

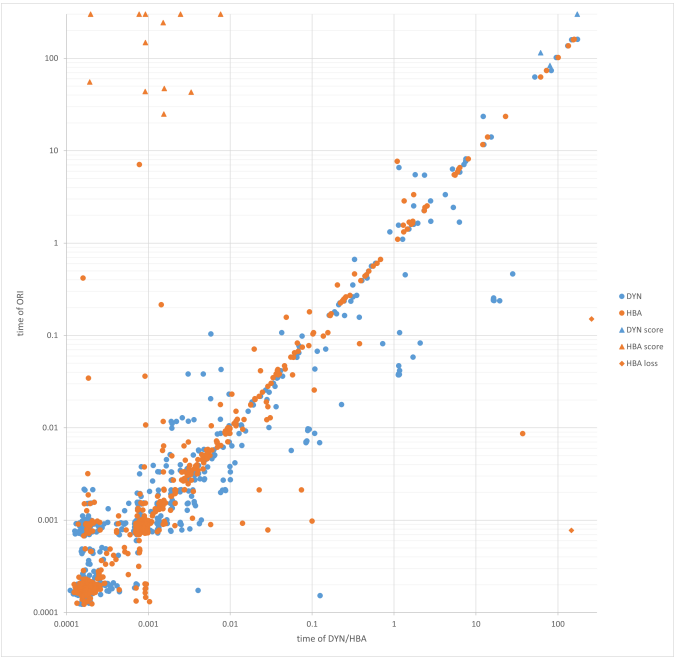
[6] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, vol. 1, no. 4, pp. 297–322, 1992.



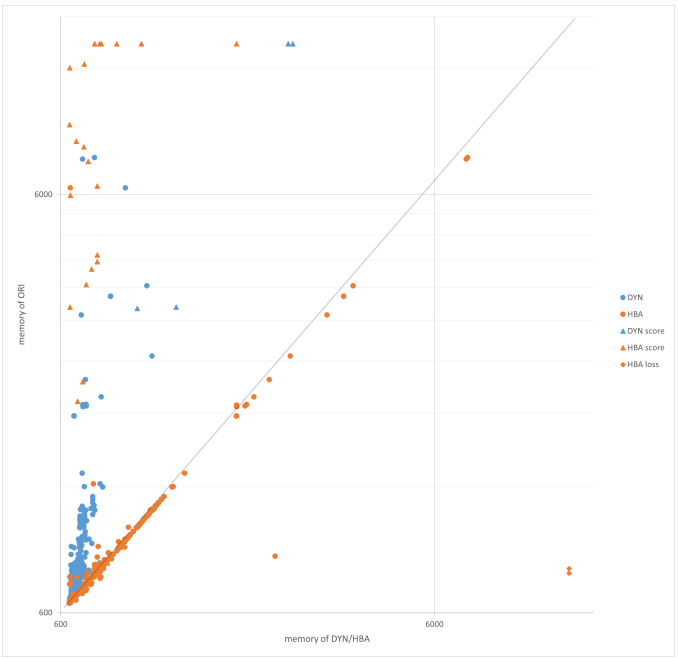
(a) DRW vs. ORI on time



(b) DRW vs. ORI on memory



(c) DYN/HBA vs. ORI on time



(d) DYN/HBA vs. ORI on memory

Fig. 5: Comparison of original method and our optimization strategies

[7] T. Liebke and C. Rosenke, "Faster enabledness-updates for the reachability graph computation." in *PNSE@ Petri Nets*, 2020, pp. 108–117.

[8] K. Wolf, "Petri net model checking with lola 2," in *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer, 2018, pp. 351–362.

[9] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[10] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer Science & Business Media, 2013.

[11] F. Kordon, "Model checking contest," https://mcc.lip6.fr/.

[12] C. He and Z. Ding, "More efficient on-the-fly verification methods of colored petri nets," *COMPUTING AND INFORMATICS*, vol. 40, no. 1, p. 195–215, Aug. 2021. [Online]. Available: https://www.cai.sk/ojs/index.php/cai/article/view/2021_1_195

[13] Z. Ding, C. He, and S. Li, "Enpac: Petri net model checking for linear temporal logic," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.12324

[14] H. Garavel, "Nested-unit petri nets: A structural means to increase efficiency and scalability of verification on elementary nets," in *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer, 2015, pp. 179–199.

[15] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, G. Ciardo, S. Dal Zilio, P. Jensen, C. He, D. Le Botlan, S. Li, A. Miner, J. Srba, and . Thierry-Mieg, "Complete Results for the 2020 Edition of the Model Checking Contest," http://mcc.lip6.fr/2020/results.php, 2020.