TAPAAL HyperLTL: A Tool for Checking Hyperproperties of Petri Nets

Bruno Maria René Gonzalez[1][0009−0006−8122−7160], Peter Gjøl Jensen[2][0000−0002−9320−9991], Stefan Schmid[1][0000−0002−7798−1711], Jiří Srba[2][0000−0001−5551−6547], and Martin Zimmermann[2][0000−0002−8038−2453]

[1] TU Berlin, Berlin, Germany
[2] Aalborg University, Aalborg, Denmark

Abstract. Petri nets are a modeling formalism capable of describing complex distributed systems and there exists a large number of both academic and industrial tools that enable automatic verification of model properties. Typical questions include reachability analysis and model checking against logics like LTL and CTL. However, these logics fall short when describing properties like non-interference and observational determinism that require simultaneous reasoning about multiple traces of the model and can thus only be expressed as hyperproperties. We introduce, to the best of our knowledge, the first HyperLTL model checker for Petri nets. The tool is fully integrated into the verification framework TAPAAL and we describe the semantics of the hyperlogic, present the tool's architecture and GUI, and evaluate the performance of the HyperLTL verification engine on a benchmark of problems originating from the computer networking domain.

Keywords: Petri nets · Model checking · HyperLTL · Tool

1 Introduction

Many important properties of systems inherently relate multiple execution traces of a system, e.g., security and information-flow properties [1,23,33,39,40,42] as well as network properties like congestion [10,24]. These are not expressible in classical specification languages like LTL [35], CTL [12], and CTL* [21], as those are restricted to reasoning about one trace at a time. Clarkson and Schneider termed properties relating multiple traces *hyperproperties* and initiated their rigorous investigation [14]. Technically speaking, a hyperproperty is a set of sets of traces, just like a trace property is a set of traces. Their study received considerable attention after the introduction of specification languages for hyperproperties, which enabled the specification, analysis, and verification of hyperproperties. The two most important ones are HyperLTL and HyperCTL* which extend LTL and CTL* by quantification over traces [13]. These logics are able to express many important hyperproperties from security like non-interference, non-inference, observational determinism, etc. [22]. On the other hand, they are also able to express properties about paths in graphs, e.g., networks, like the existence of several disjoint paths between a source and a target node. This allows us to formalize quantitative aspects like congestion using HyperLTL as a requirement on the maximal number of flows that can traverse any given edge.

Petri nets [34] are widely used to represent concurrent and distributed systems due to their expressive power and an intuitive graphical representation. Despite the versatility of Petri nets, no prior HyperLTL verification tool has provided user-friendly support for designing and verifying Petri net models. Existing approaches often rely on textual specifications or lack intuitive interfaces.

To address this challenge, we introduce TAPAAL HyperLTL, a novel HyperLTL model checker integrated into the TAPAAL [17] verification suite, specifically designed to verify complex temporal properties of distributed systems modeled as Petri nets. Our implementation is the first to bring HyperLTL verification to Petri nets, offering a robust verification engine coupled with an intuitive user interface for modeling as well as debugging purposes.

To evaluate our tool, we conduct an extensive case study showing the applicability of HyperLTL for the analysis of congestion in a computer networking setting. Our results show that our HyperLTL engine outperforms the baseline approach based on self-composition [2,39] and achieves competitive performance compared to state-of-the-art tools like MCHyper [22].

Related work. HyperLTL and its branching-time companion HyperCTL* have been introduced and their model-checking problems have been shown decidable in the seminal work of Clarkson et al. [13]. In general, model checking of HyperLTL is TOWER-complete [36,32] in the number of quantifier alternations. Hence, almost all tool development has been concerned with the alternation-free fragment, although recently the first tools tackling (a small number of) alternations have been presented.

For example, the tool MCHyper models the system using And-Inverter Graphs (AIGs) and has originally been restricted to alternation-free formulae [22] (like our tool), where it relies on the ABC [9] backend. More recently, it has been extended to handle one alternation using a game-based approach [15]. On the other hand, the tool AutoHyper handles quantifier alternations [5] by implementing an automata-theoretic model checking algorithm, relying on efficient automata inclusion checking.

Another approach for handling the inherent complexity of HyperLTL model checking is to consider incomplete methods like bounded model checking, which searches for counterexamples of bounded size. Hsu et al. [27,26] implemented this in their tool HyperQube using a reduction to QBF.

Finally, model checking asynchronous extensions of HyperLTL has been studied by Baumeister et al. [3] and probabilistic extensions by Dode et al. [18]. Most recently, the game-based approach mentioned above has been generalized [4,41] and planning-based [6] algorithms and implementations have been presented.

None of the existing tools mentioned above can handle Petri nets natively. Thus, our tool offers an alternative modeling language based on Petri nets, which naturally support concurrency, while existing tools use NuSMV [11] models (like HyperQube) or VHDL [37] and VeriLog [38] models (like MCHyper).

2 Modeling Formalism and HyperLTL Logic

We shall now semi-formally introduce the Petri net model as well as the syntax and semantics of the variant of HyperLTL that is supported by our tool and tailored to express properties of Petri nets.

2.1 Petri nets

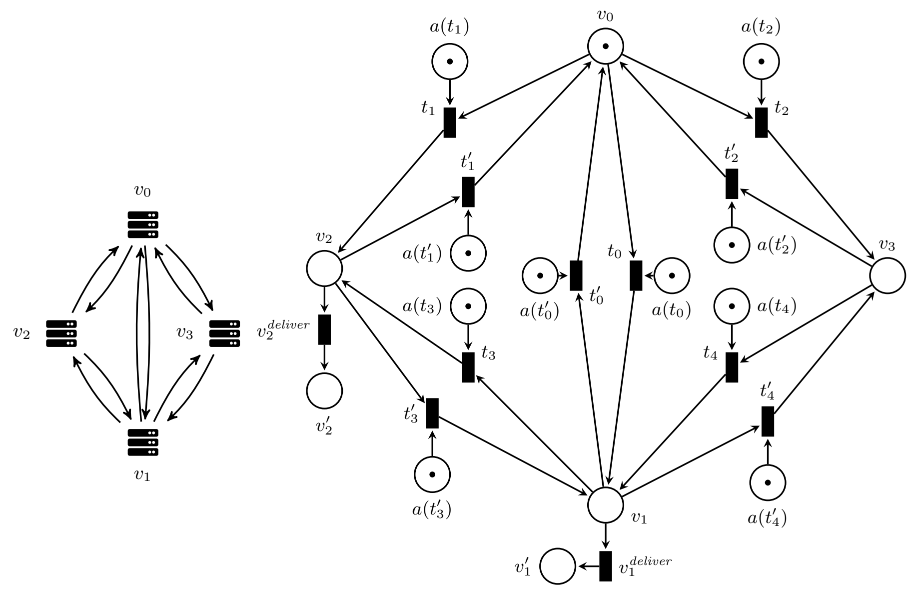
Our tool uses the classical Petri net (PN) model [34] with weighted and inhibitor arcs. It also supports colored Petri nets, following the PNML syntax used in the annual Model Checking Contest (MCC) [31]. The colored PNs are unfolded into classical P/T (place/transition) nets, after which the HyperLTL model checking is executed.

Figure 1b shows an example of a P/T net where places from the set $P = \{v_0, \ldots, v_3, a(t_0), a(t'_0), \ldots, a(t_4), a(t'_4)\}$ are drawn as circles, transitions from the set $T = \{t_1, t'_1, \ldots, t_4, t'_4, v_1^{deliver}, v_2^{deliver}\}$ are drawn as rectangles, and arcs are the directed edges connecting either places to transitions or transitions to places. Unless otherwise stated, the default weight of all arcs is 1.

A *marking* $M\colon P \to \mathbb{N}^0$ is a function that represents the placement of tokens (denoted as dots) across the places in the net. A transition t is *enabled* in a marking M if there are enough tokens in all of the input places to the transition. An enabled transition t can *fire* and produce the new marking M', written as $M[t\rangle M'$, by (i) removing as many tokens from the input places as is the weight of the corresponding arc, and (ii) producing new tokens to every output place of the transition, again according to the weights of the output arcs. For example, firing the transition t_1 in Figure 1b removes the tokens from v_0 and $a(t_1)$ and adds a token to v_2. All other tokens are unchanged.

The Petri net in Figure 1b models all possible routing sequences for the computer network depicted in Figure 1a where a packet (token) starts at the node v_0 and the aim is to reach the node v_1 or v_2. Moreover, every link in the network corresponds to some transition t in the Petri net and once this transition fires, the token in the place $a(t)$ is consumed, representing the fact that the corresponding link is now occupied.

A *trace* (run) in a Petri net is an infinite sequence $\rho = M_0 M_1 M_2 \cdots$ of markings such that for every $n \geq 0$ either (i) $M_n[t_n\rangle M_{n+1}$ for some transition $t_n \in T$, or (ii) $M_{n+1} = M_n$ in case that M_n is a deadlock, i.e., if M_n does not enable any transition. As HyperLTL is interpreted over infinite traces, we introduce the stuttering to prolong possibly deadlocked traces into infinite ones (as it is e.g., assumed in the MCC [31]). Given a trace $\rho = M_0 M_1 M_2 \cdots$, we denote the n'th marking M_n in the trace by ρ^n.



(a) A comp. network (b) A Petri net N modeling the computer network

$$\varphi_1 \equiv \exists\pi_1.\ \exists\pi_2.\ (\mathbf{F}\,\pi_1.v_1' = 1) \wedge (\mathbf{F}\,\pi_2.v_1' = 1) \wedge \mathbf{G}\,\mathit{noCongestion2}$$
$$\varphi_2 \equiv \exists\pi_1.\ \exists\pi_2.\ \exists\pi_3.\ (\mathbf{F}\,\pi_1.v_1' = 1) \wedge (\mathbf{F}\,\pi_2.v_1' = 1) \wedge (\mathbf{F}\,\pi_3.v_1' = 1) \wedge \mathbf{G}\,\mathit{noCongestion3}$$
$$\varphi_3 \equiv \exists\pi_1.\ \exists\pi_2.\ (\mathbf{F}\,\pi_1.v_2' = 1) \wedge (\mathbf{F}\,\pi_2.v_2' = 1) \wedge \mathbf{G}\,\mathit{noCongestion2}$$
$$\varphi_4 \equiv \exists\pi_1.\ \exists\pi_2.\ \exists\pi_3.\ (\mathbf{F}\,\pi_1.v_2' = 1) \wedge (\mathbf{F}\,\pi_2.v_2' = 1) \wedge (\mathbf{F}\,\pi_3.v_2' = 1) \wedge \mathbf{G}\,\mathit{noCongestion3}$$

where

$$\mathit{noCongestion2} \equiv \bigwedge_{t \in T \smallsetminus \{v_1^{deliver}, v_2^{deliver}\}} (\pi_1.a(t) + \pi_2.a(t) \geq 1)$$
$$\mathit{noCongestion3} \equiv \bigwedge_{t \in T \smallsetminus \{v_1^{deliver}, v_2^{deliver}\}} (\pi_1.a(t) + \pi_2.a(t) + \pi_3.a(t) \geq 2)$$

(c) Examples of HyperLTL formulae where $N \models \varphi_1$, $N \models \varphi_2$, $N \models \varphi_3$ and $N \not\models \varphi_4$

Fig. 1: Example of a Petri net and HyperLTL formulae

2.2 HyperLTL

HyperLTL [13] extends LTL [35] (which is evaluated over single traces) by quantification over multiple traces, and is therefore evaluated over sets of traces. Our tool supports an alternation-free hyperlogic specifically tailored to Petri nets.

Figure 1c shows example formulae in the logic where the π_i are trace variables ranging over infinite traces of a Petri net. Every formula starts with either existential or universal quantification over a list of trace variables $\pi_1, \ldots, \pi_m$. Thus, quantification assigns traces of the Petri net to the trace variables. This is followed by a formula composed of the standard LTL temporal operators that is (synchronously) evaluated over the quantified traces:

- $\mathbf{X}\,\varphi$ stating that φ holds at the next position,
- $\mathbf{F}\,\varphi$ stating that φ holds at some future position,
- $\mathbf{G}\,\varphi$ stating that φ holds at all future positions, and
- $\psi\,\mathbf{U}\,\varphi$ stating that φ holds at some future position and ψ holds at all intermediate positions.

Finally, we consider two types of atomic propositions:

- for every variable π and every transition t there is a proposition $\pi.\mathtt{en}_t$, and
- we allow linear (in)equalities of the form $\sum_\ell c_\ell \cdot \pi_{i_\ell}.p_\ell \bowtie b$ where the c_ℓ and b are integer constants, the π_{i_ℓ} are trace variables, and the p_ℓ are places of the net, and where $\bowtie \in \{<, \leq, =, \geq, >\}$ is a comparison operator.

Now, assuming that the trace ρ_i is assigned to the variable π_i for each i, we evaluate atomic propositions at position n as follows:

- $\pi_i.\mathtt{en}_t$ is satisfied if t is enabled in the n'th marking ρ_i^n, and
- $\sum_\ell c_\ell \cdot_{i_\ell} .p_\ell \bowtie b$ is satisfied if the inequality obtained by replacing each $\pi_{i_\ell}.p_\ell$ with the number of tokens in the marking $\rho_{i_\ell}^n$, i.e., the value $\rho_{i_\ell}^n(p_\ell)$, is valid.

For a formal definition of the syntax and semantics of HyperLTL, see, e.g., [13].

Coming back to our example in Figure 1, the formula φ_1 (resp. φ_3) from Figure 1c expresses that there are two traces that both eventually (but possibly at different positions) receive a token in v_1' (resp. v_2') and each transition t is fired in at most one of the traces (hence there must be a token in $a(t)$ in at least one of the two traces). In other words, φ_1 and φ_3 express that there are two disjoint paths in the graph in Figure 1a, starting at v_0 and leading to v_1 resp. v_2. Analogously, φ_2 and φ_4 express similar properties, but requiring the existence of three disjoint paths. Hence, φ_1, φ_2, and φ_3 are satisfied by the example net N, but not φ_4.

The reason for introducing the places v_1' and v_2' is that once the token initially in v_0 arrives to one of them, the corresponding trace gets stuck and the last reached marking is allowed to stutter and hence does not use any of the remaining link capacities. This is important as the existentially quantified traces may be of different lengths before reaching the goal place, and such traces must globally synchronize.

3 Tool Implementation and Graphical User Interface

The verification engine of our tool is implemented in C++ and extends the existing LTL verification engine that is part of the `verifypn` command line tool [28]. The HyperLTL engine supports nets described in PNML [7] and, for universal formulae φ, constructs in an on-the-fly manner the vector of markings currently reached in all the considered traces and explores its product with the Büchi automaton representing the negation of the LTL property obtained by dropping the quantifiers of φ. On this product Büchi automaton, we perform a classical search for a reachable accepting loop using the nested DFS search

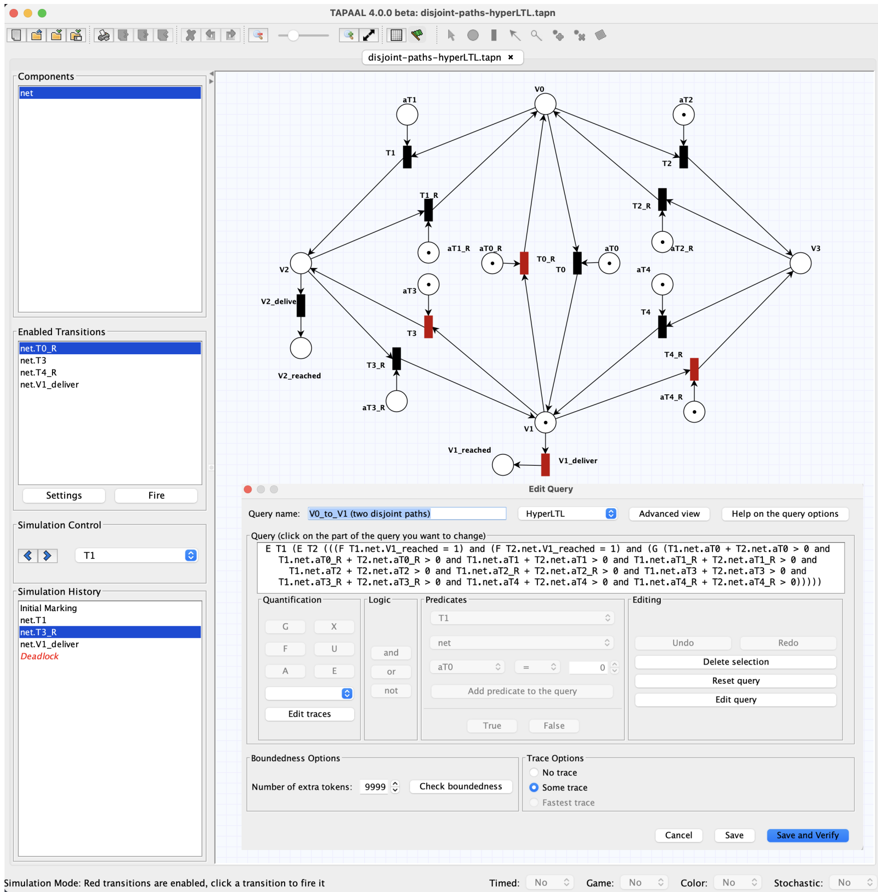


Fig. 2: TAPAAL HyperLTL screenshot (simulator mode with a query dialog)

strategy [16]. If such a loop is found, the engine returns the verification answer together with the set of traces (in an XML format) that form such a loop. In the case where no counter-example exists, the tool reports that the property is satisfied together with statistics about the explored state-space. Existential quantification is handled by negating the formula and swapping the results the tool reports.

The HyperLTL engine is directly called from the tool TAPAAL [17] that has been extended with a graphical way to construct HyperLTL queries as well as a simulator that allows to replay multiple traces returned by the engine. Figure 2 displays a screenshot of the TAPAAL HyperLTL interface. The GUI is in simulation mode where the user can select the traces returned by the verification engine (currently, trace T1 is selected) and simulate the traces in the GUI. A graphical dialog for creating HyperLTL queries is shown as overlay. The

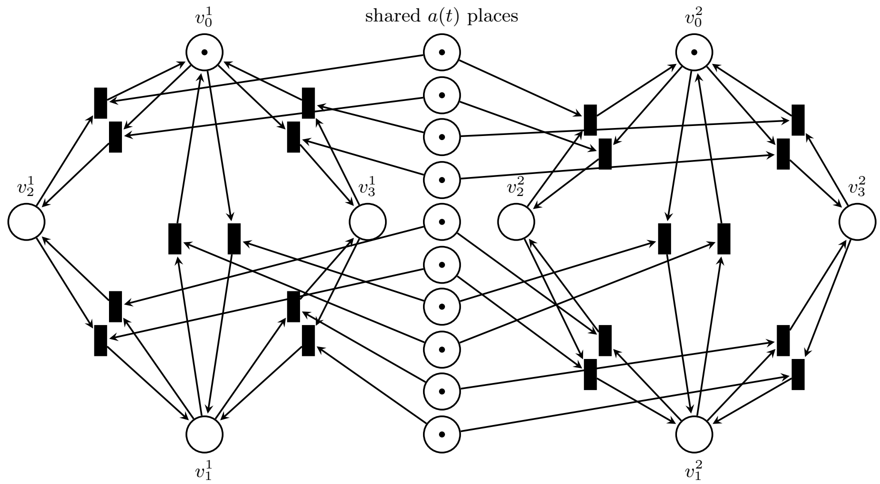


Fig. 3: Self-composition with the LTL query $(\mathbf{F}\, v_1^1 = 1) \wedge (\mathbf{F}\, v_1^2 = 1)$

tool is available at `http://www.tapaal.net/downloads`, including a complete reproducibility package [25].

4 Performance Evaluation and Case Study

We evaluate the performance of our HyperLTL model checker on a case study inspired by routing problems from computer networking [19]. For given source and destination nodes s and t, and for a given network (directed graph), we want to find k directed paths from s to t such that these paths do not cause congestion on any of the links (edges) in the network. In our simplified scenario, we say that an edge is congested if there are strictly more than ℓ paths from s to t that use the given edge; hence ℓ indirectly models edge capacities. The introductory example in Figure 1b shows how this problem can be modeled as a Petri net. The formulae for $k = 2, 3$ and $\ell = 1$ are depicted in Figure 1c as φ_1 and φ_3 for the target node v_1 and as φ_2 and φ_4 for the target node v_2.

Our benchmark contains 3900 HyperLTL formulae, evaluated on Petri net models of 260 real-world network topologies from the Topology Zoo dataset [30]. We consider three (k, ℓ) problem variants for $(2, 1)$, $(3, 1)$ and $(4, 2)$, where for each network topology we generate five HyperLTL queries for randomly selected pairs of source and target nodes. To balance the number of true and false queries in the benchmark, the source is selected to be a random high-degree node. The experiments are executed on an AMD EPYC 7551 processor running at 1996 MHz, with 900 seconds timeout. TAPAAL additionally had a 2GB memory limit.

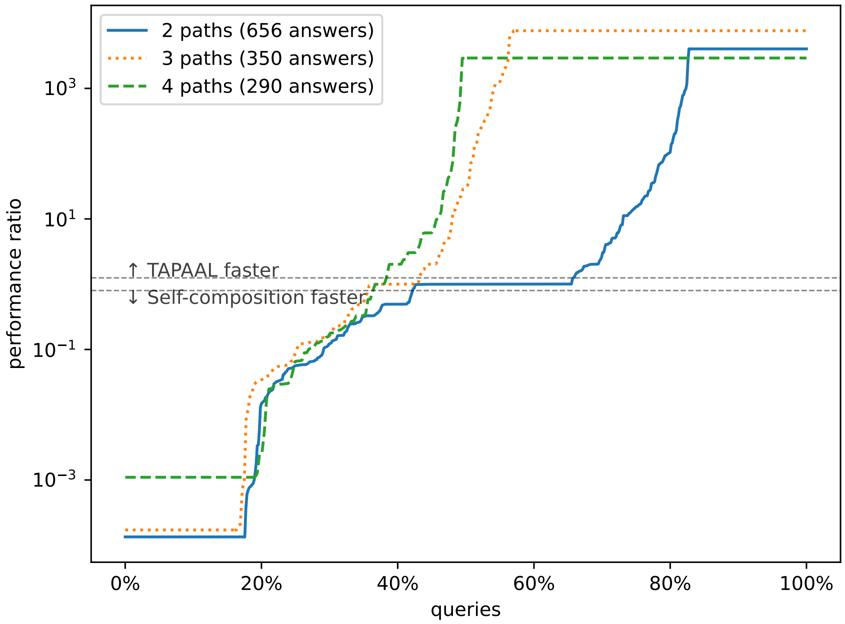


Fig. 4: Ratio plot of TAPAAL HyperLTL vs. Self-composition

First, we compare our HyperLTL implementation (referred to as TAPAAL) with a self-composition approach [2,39] that creates a copy of the composed model for each trace and adds a synchronization mechanism to the model in order to guarantee that we iteratively perform one step in each copy of the model before we evaluate the predicates and continue with another single step in each copy. This allows us to reduce the HyperLTL formula into a normal LTL formula where instead of each trace we now refer to the respective copy of the model. However, this is at the expense of creating a possibly complicated model that explodes with the number of traces and additionally implements a synchronization mechanism in order to keep all copies synchronized.

In our concrete example, the self-composition does not require such a complicated synchronization mechanism as for each quantified trace we can create a copy of the net that can run completely concurrently (avoding the lock-step synchronization), while checking for the congestion using the shared places $a(t)$ that contain as many tokens as is the edge capacity. Figure 3 shows our simplified self-composition for our running example as well as a classical LTL query that expresses the same property as the HyperLTL formula φ_1 from Figure 1c. To verify the classical LTL formula on the self-composed system, we benchmark our tool against the LTL engine of TAPAAL [29], the winner in the 2025 Model Checking Contest [31] in the LTL category. A ratio plot is depicted in Figure 4 where the x-axis contains all queries solved by at least one of the methods,

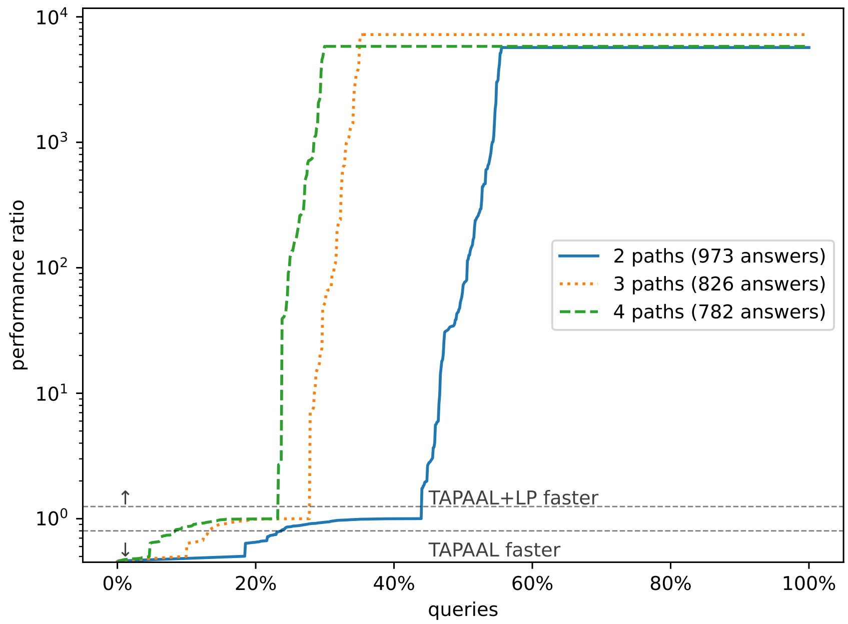


Fig. 5: Ratio plot of TAPAAL HyperLTL with and without LP Check

sorted by the ratio of self-composition running time divided by TAPAAL running time. We remark that because the plot contains three different problem instances where the tools solve different number of queries (the numbers of answers in the parenthesis show the total number of solved problems by at least one of the methods), we use the percentage scale on the x-axis instead of the absolute count. For two disjoint paths, both methods are comparable, however, for 3 and 4 paths there is a clear advantage of using our new HyperLTL implementation. For example, for 4 paths, the self-composition timeouts (depicted by the straight horizontal line) on more than 50% of all queries that the HyperLTL implementation managed to solve. This is in particular true for queries with positive answers, as the on-the-fly method that we implemented in TAPAAL is more efficient than self-composition, where the net size explodes with number of trace variables in the HyperLTL formula.

In order to further improve the performance of our tool on negative queries, we employ an over-approximation method based on state-equations and linear programming [8] (we refer to this method as TAPAAL+LP). We create the self-composition net and apply the fast LP check that can in many cases show that a HyperLTL query is not satisfied, notably without performing any state-space search. If the LP check is inconclusive, we run our HyperLTL engine to perform the state-space exploration using nested depth-first search. Figure 5 shows the ratio plot of TAPAAL vs. TAPAAL+LP. Most of the additionally answered

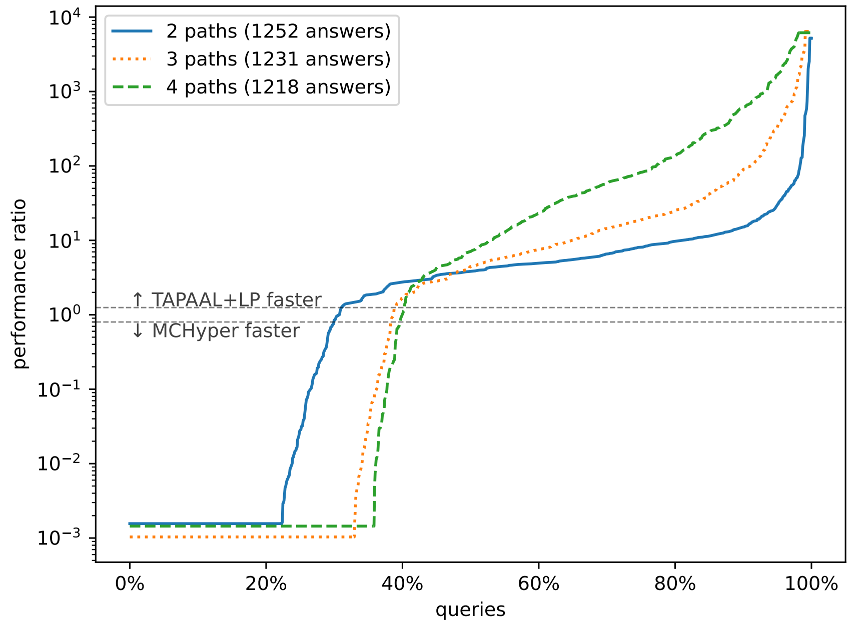


Fig. 6: Ratio plot of TAPAAL HyperLTL+LP vs. MCHyper

queries are negative ones that are solved using this LP check. Even though the LP check is in general often beneficial, the tool allows the user to skip it and proceed directly to the state-space search if needed.

Finally, we compare our HyperLTL engine with the LP check against MCHyper [15] which is a state-of-the-art model checker for HyperLTL properties. MCHyper operates by encoding the system, described as an AIG circuit, and the formula into a new compact circuit of linear size w.r.t. to the size of the model and the formula [22]. While Petri nets can naturally represent nonnegative integers as a number of tokens in places, in MCHyper these numbers have to be encoded into Boolean variables. To this end, we translated all 260 network topologies in our benchmark into AIG circuits, using a unary encoding for the nodes of the topologies. The transitions of the system are modeled as a simple state machine, where each transition t additionally requires a latch, representing the token in the $a(t)$ place, to be enabled. To allow traces of differing lengths, the circuit cannot leave the target state after entering it (similarly to the Petri net encoding). The formula is translated into the MCHyper format. For a given problem with parameters (k, ℓ), we encode the sum by simply enumerating the $\binom{k}{\ell}$ terms, as the values are sufficiently small. The comparison of TAPAAL HyperLTL+LP vs. MCHyper is provided in Figure 6. It shows that our tool is faster on about 60% of all queries, however, MCHyper solves a significant number of queries where our tool timeouts. This is caused by the fact that for alternation-free formulae (like in our benchmarks), the encoding of MCHyper allows the property to be verified by a simpler reachability query on the circuit. This enables it to rely on the specialized verification tool ABC [9], which implements state-of-the-art SAT-solvers including PDR (property-directed reachability heuristics) [20].

5 Conclusion

We presented the first HyperLTL verification engine for Petri nets, implemented a GUI that allows the user to visually design Petri net models as well as HyperLTL queries and provides debugging feedback as the traces discovered by our engine can be simulated in the TAPAAL GUI. We showed that our HyperLTL engine is more efficient than an alternative self-composition approach and that it is competitive with the state-of-the-art HyperLTL model checker MCHyper. In future work, we plan to transfer the techniques that enable MCHyper to quickly answer positive HyperLTL queries, in particular property-directed search heuristics, into TAPAAL, in order to further improve its performance.

Acknowledgements. Research supported by the German Research Foundation (DFG), project ReNO (SPP 2378), 2023–2027, and by DIREC—Digital Research Centre Denmark.

References

1. Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.
2. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011.
3. Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *CAV 2021, Part I*, volume 12759 of *LNCS*, pages 694–717. Springer, 2021.
4. Raven Beutner and Bernd Finkbeiner. Prophecy variables for hyperproperty verification. In *CSF 2022*, pages 471–485, , 2022. IEEE.
5. Raven Beutner and Bernd Finkbeiner. AutoHyper: Explicit-state model checking for HyperLTL. In *TACAS 2023, Part I*, volume 13993 of *LNCS*, pages 145–163. Springer, 2023.
6. Raven Beutner and Bernd Finkbeiner. Non-deterministic planning for hyperproperty verification. In *ICAPS 2024*, pages 25–30. AAAI Press, 2024.
7. Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri net markup language: Concepts, technology, and tools. In *ICATPN 2003*, volume 2679 of *LNCS*, pages 483–505. Springer, 2003.
8. F.M. Boenneland, J. Dyhr, P.G. Jensen, M. Johannsen, and J. Srba. Simplification of CTL formulae for efficient model checking of Petri nets. In *Petri Nets 2018*, volume 10877 of *LNCS*, pages 143–163. Springer-Verlag, 2018.
9. Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV 2010*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010.

10. Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic engineering with equal-cost-multipath: An algorithmic perspective. *IEEE/ACM Trans. Netw.*, 25(2):779–792, 2017.
11. Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV 2002*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.
12. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
13. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *POST 2014*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
14. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.
15. Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *CAV 2019 Part I*, volume 11561 of *LNCS*, pages 121–139. Springer, 2019.
16. Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods Syst. Des.*, 1(2/3):275–288, 1992.
17. Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jirí Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *TACAS 2012*, volume 7214 of *LNCS*, pages 492–497. Springer, 2012.
18. Oyendrila Dobe, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. Hyperprob: A model checker for probabilistic hyperproperties. In *FM 2021*, volume 13047 of *LNCS*, pages 657–666. Springer, 2021.
19. D Anthony Dunn, Wayne D Grover, and Mike H MacGregor. Comparison of k-shortest paths and maximum flow routing for network facility restoration. *IEEE Journal on selected areas in Communications*, 12(1):88–99, 1994.
20. Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *FMCAD 2011*, pages 125–134. FMCAD Inc., 2011.
21. E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.
22. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *CAV 2015, Part I*, volume 9206 of *LNCS*, pages 30–48. Springer, 2015.
23. Riccardo Focardi and Roberto Gorrieri. Classification of security properties (part I: information flow). In *Foundations of Security Analysis and Design, Tutorial Lectures [revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design, FOSAD 2000]*, volume 2171 of *LNCS*, pages 331–396. Springer, 2000.
24. Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing OSPF weights. In *INFOCOM 2000*, pages 519–528. IEEE Computer Society, 2000.
25. Bruno Maria René Gonzalez, Peter Gjøl Jensen, Stefan Schmid, Jiri Srba, and Martin Zimmermann. Reproducibility package for "TAPAAL HyperLTL: A tool for checking hyperproperties of Petri net", July 2025. `https://doi.org/10.5281/zenodo.15854993`.

26. Tzu-Han Hsu, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. Bounded model checking for asynchronous hyperproperties. In *TACAS 2023, Part I*, volume 13993 of *LNCS*, pages 29–46. Springer, 2023.
27. Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In *TACAS 2021, Part I*, volume 12651 of *LNCS*, pages 94–112. Springer, 2021.
28. Jonas Finnemann Jensen, Thomas Nielsen, Lars Kaerlund Oestergaard, and Jirí Srba. TAPAAL and reachability analysis of P/T nets. *Trans. Petri Nets Other Model. Concurr.*, 11:307–318, 2016.
29. P.G. Jensen, J. Srba, N.J. Ulrik, and S.M. Virenfeldt. Automata-driven partial order reduction and guided search for ltl model checking. In *Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'22)*, volume 13182 of *LNCS*, pages 151–173. Springer-Verlag, 2022.
30. Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.
31. F. Kordon, F. Hulin-Hubard, L. Jezequel, E. Paviot-Adet, Q. Nivon, N. Amat., B. Berthomieu, S. Dal Zilio, , Z. Ding, Y. He, S. Li, C. Jiang, P. G. Jensen, J. Srba, and Y. Thierry-Mieg. Complete Results for the 2025 Edition of the Model Checking Contest. `https://mcc.lip6.fr/2025/results.php`, 2025.
32. Corto Mascle and Martin Zimmermann. The keys to decidable HyperLTL satisfiability: Small models or very simple formulas. In *CSL 2020*, volume 152 of *LIPIcs*, pages 29:1–29:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
33. John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. Software Eng.*, 22(1):53–67, 1996.
34. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt, 1962.
35. Amir Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57. IEEE, Oct 1977.
36. Markus N. Rabe. *A temporal logic approach to Information-flow control*. PhD thesis, Saarland University, 2016.
37. IEEE Computer Society. IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, pages 1–218, 1988.
38. IEEE Computer Society. IEEE standard for Verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.
39. Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS 2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
40. Dennis M. Volpano. Safety versus secrecy. In Agostino Cortesi and Gilberto Filé, editors, *SAS 1999*, volume 1694 of *LNCS*, pages 303–311. Springer, 1999.
41. Sarah Winter and Martin Zimmermann. Prophecies all the way: Game-based model-checking for HyperQPTL beyond $\forall^*\exists^*$. *arXiv*, 2504.08575, 2025. Accepted for publication at CONCUR 2025.
42. Aris Zakinthinos and E. Stewart Lee. A general theory of security properties. In *SCP 1997*, pages 94–102. IEEE Computer Society, 1997.