Towards Safe Agents with Graph Rewriting

Adam Vandervorst and Anneline Daggelinckx*

Qoba.ai
{adam,anneline}@qoba.ai

Overview Formal methods are ubiquitous in computer science, but they have yet to impact the study of intelligent agents significantly. While various logics like temporal logics [6, 15] (guarantees on action sequencing), Markov automata [2] (probabilities over reachability), and hypernets [22] (observational equivalence) can describe parts of agent behavior, they are hard to combine and have a significant specialization overhead. As AI systems scale up, our methods for modeling and control need to do so as well by providing a transparent and interactive environment for growing a proof base and world model using a blanket of methods on a scalable substrate. We propose to unify specialized logic methods using a powerful graph rewriting system. For our proof of concept[1], we show the direct enrichment of the world model by semantic reasoning, the direct simplification of LTL formulae, and a meta-rewriting approach that checks the formulae on model paths. We see further uses in autonomous enriching and simplification, as well as in machine-human and human-human collaborative editing, all backed by a massive parallel backend.

Context There are many properties of an AI agent that we value when it is acting in the world. Naturally, the desire to capture and enforce properties formally has been expressed [3,26, 27]. Both from a practical standpoint (concerning compute resources and partial observation) and a theoretical standpoint (concerning being decidable), it is mandatory to make multiple projections of the systems of interest and use a blanket of formal methods to cover the desired safety aspects. There is a large body of foundational work on verifying and enforcing properties, yet simultaneously capturing multiple aspects of a domain is reserved for the most formal systems like programming languages under the K Framework [20]. In this work, we focus on properties that state-of-the-art methods are able to assert from domain-specific systems, as these tend to scale best.

Graph rewriting has been used successfully in various relevant domains:

- Rewriting graph representations of queries for scalability [14]
- Rewriting engineering domains to be more amendable to analysis [13]
- Rewriting ASTs for reasoning about programming languages in a uniform framework [11]
- Rewriting real-world networks to increase robustness [8]
- Merging nodes to summarize graphs [19]
- Rewriting hypergraphs as the main mode of computation on knowledge representation [29]

One aspect where related initiatives have trouble scaling is rule creation, which we aim to address with transparent checking (e.g., rewrite rules can be stepped through, making vacuity detection [16] easier) and long-lived rule building (utilizing an approach similar to theories in physics [30]). Another bottleneck to scaling existing logical methods is their sequential implementation, and reformulating them as a system of graph rewrites or graph grammar [7] may help.

*Authors contributed equally

[1]Our open-source Scala implementation is available September 2024 at https://github.com/Qoba-AI/AITP-24

Approach We propose hosting multiple successful approaches on a shared substrate, with programmatic projections and recombinations of the ground truth data. The world model, rules, results, and meta-information are stored in the same graph and used in some of the same rewrites. Throughout the program's long-lived operation, layers of properties and abstraction are built up and version-controlled.

Many world models can be represented as graphs. A trivial example is discrete world models with their transition systems, but any domain abstraction that admits to an algebraic decomposition can be embedded straightforwardly. We propose to work directly on these graph models using a fine form of graph rewriting called Patch Graph Rewriting (PGR) [23]. PGR is a uniform graphical rewriting language that allows to specify how the replacement graph should be connected to the host graph. While generalizations of graph grammars [7] and DPO [5] have been proposed, PGR gives more control and has better legibility to collaborators in the AI field (Appendix A).

For illustrative purposes, we can directly mask an agent action we match as harmful using a seemingly direct change to the world model (Appendix B). Besides viewing the graph modulo a criterion (by merging nodes [19]) or working with more abstract states to reduce size, abstract states can summarize safe or unsafe areas to preferably avoid (Appendix D). Safety measures in the constructed controller can also include consulting a human before executing certain (e.g., irreversible) actions (Appendix C).

We can also rewrite the rules themselves. Linear Temporal Logic (LTL) is a modal logic that expresses time-related safety properties over infinite paths. We check whether an LTL formula holds in a Kripke model by translating the LTL formula to a Büchi automaton [10]. The constructed automata are far from optimal in size, and we can increase interpretability and verification speed by applying a set of reducing PGR rules until a fixed point is reached (Appendix E).

Key to real-world effectiveness is the ability to import formal descriptions and update the knowledge with agent observations. In an exploration scenario, we can continually add discovered information to the graph and efficiently update the tower of properties by only checking the possibly affected rewrites.

For deployment, multiple modes of operation may be combined to provide guarantees at different levels. For example, in a runtime assurance setting [21], we can specify conditions in LTL$_f$ [4], a variant of LTL for finite paths like the plans proposed by a black box. Using the simplified Büchi automaton of an LTL$_f$ formula, we can use PGR rules to traverse a path (runtime information) and the Büchi automaton (derived from another rule) simultaneously, checking whether it holds (Appendix F).

Outlook Generalizing LTL formula generation and simplification to CTL* [6] and the μ-calculus [15] is an obvious next step. Matching logic [24] would be powerful in combination with structural states (where composite properties live in the graph, too). Reachability rules [25] can contain predicates that (potentially recursively) depend on big or small step reductions. Adding a similar predication on patch graph rewrites would require irrealis rewriting contexts or pre-processing steps.

Many real-world applications have a notion of feasibility on top of reachability. This comes in the form of the probability of success [17, 18] and resources to reach the goal [1, 31]. We believe a generalization of PGR can accommodate both notions of grading [9] simultaneously. By providing a lattice on labels that can be matched on, as well as allowing the joining of labels in the rule's RHS, both resources and likelihood can be predicated on and propagated throughout the graph (similar to Information Programming [28]).

References

[1] Natasha Alechina, Brian Logan, Hoang Nga Nguyen, and Franco Raimondi. Decidable model-checking for a resource logic with production of resources. In *ECAI*, volume 14, pages 9–14, 2014.

[2] Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns. A modest approach to markov automata. *ACM Trans. Model. Comput. Simul.*, 31(3), aug 2021.

[3] David Dalrymple. Safeguarded ai: Constructing guaranteed safety. Programme thesis, ARIA, UK, 2024. Available at https://www.aria.org.uk.

[4] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 854–860. AAAI Press, 2013.

[5] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 167–180, 1973.

[6] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[7] H. Fahmy and D. Blostein. A survey of graph grammars: theory and applications. In *11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*, volume 1, pages 294,295,296,297,298, Los Alamitos, CA, USA, sep 1992. IEEE Computer Society.

[8] Scott Freitas, Diyi Yang, Srijan Kumar, Hanghang Tong, and Duen Horng Chau. Graph vulnerability and robustness: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5915–5934, 2023.

[9] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, and Tetsuya Sato. Graded hoare logic and its categorical semantics. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 234–263, Cham, 2021. Springer International Publishing.

[10] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Conference on Protocol Specification, Testing and Verification*, pages 3–18. Springer, 1995.

[11] Dan R. Ghica. Operational semantics with hierarchical abstract syntax graphs. *Electronic Proceedings in Theoretical Computer Science*, 334:1–10, February 2021.

[12] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Cautious reinforcement learning with logical constraints. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 483–491, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

[13] Lothar Kolbeck, Simon Vilgertshofer, Jimmy Abualdenien, and André Borrmann. Graph rewriting techniques in engineering design. *Frontiers in Built Environment*, 7, February 2022.

[14] George Konstantinidis and José Luis Ambite. Scalable query rewriting: a graph-based approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 97–108, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

[16] Orna Kupferman and Moshe Y Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, 4:224–233, 2003.

[17] Iain Little, Douglas Aberdeen, Sylvie Thiébaux, et al. Prottle: A probabilistic temporal planner. 2005.

[18] Iain Little, Sylvie Thiebaux, et al. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, pages 1–10, 2007.

[19] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and

applications: A survey. *ACM Computing Surveys*, 51(3):1–34, June 2018.

[20] Dorel Lucanu, Traian Florin ŞerbănuŢă, and Grigore Roşu. 𝕂 framework distilled. In Franciso Durán, editor, *Rewriting Logic and Its Applications*, pages 31–53, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[21] Usama Mehmood, Sanaz Sheikhi, Stanley Bak, Scott A. Smolka, and Scott D. Stoller. The black-box simplex architecture for runtime assurance of autonomous cps. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 231–250, Cham, 2022. Springer International Publishing.

[22] Koko Muroya. Hypernet semantics and robust observational equivalence. In *Proceedings of the Coalgebraic Methods in Computer Science 2020 (CMCS 2020)*. Coalgebraic Methods in Computer Science, October 2020. Invited talk at CMCS 2020 Online.

[23] Roy Overbeek and Jörg Endrullis. Patch graph rewriting. In *Graph Transformation: 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25–26, 2020, Proceedings 13*, pages 128–145. Springer, 2020.

[24] Grigore Rosu. Matching logic. *CoRR*, abs/1705.06312, 2017.

[25] Grigore Rosu, Andrei Stefanescu, Stefan Ciobâca, and Brandon Moore. Reachability logic. 2012.

[26] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Toward verified artificial intelligence. *Commun. ACM*, 65(7):46–55, jun 2022.

[27] Max Tegmark and Steve Omohundro. Provably safe systems: the only path to controllable agi, 2023.

[28] Adam Vandervorst. Information Programming. https://adamv.be/Information-programming, 2021. [Online; accessed 10-May-2024].

[29] Linas Vepstas, Ben Goertzel, et al. OpenCog AtomSpace. https://github.com/opencog/atomspace/, 2008. [Online; accessed 10-May-2024].

[30] Tailin Wu and Max Tegmark. Toward an artificial intelligence physicist for unsupervised learning. *Physical Review E*, 100(3), September 2019.

[31] Chanyeol Yoo, Robert Fitch, and Salah Sukkarieh. Probabilistic temporal logic for motion planning with resource threshold constraints. In *Robotics: Science and Systems*, 2012.

A Patch Graph Rewriting

In graph rewriting, rewrite rules express how to substitute parts of a graph with other subgraphs. Let G be a graph. A rewrite rule $L \to R$ denotes that any subgraph of G isomorphic to the pattern graph L should be replaced by the replacement graph R. When matching a pattern graph P to a subgraph of G, G can be divided into a match graph M (the subgraph of G isomorphic to pattern P), context C (the biggest subgraph of G that does not contain M) and a patch J (all edges that are not in M or C).

Patch Graph Rewriting (PGR) is a graphical rewriting language that, unlike classic graph rewriting frameworks like single or double pushout graph rewriting (SPO or DPO), allows one to specify how to connect the match to the context. We demonstrate this with figures from R. Overbeek and J. Endrullis [23] [2].

Figure 2 shows an example of a PGR rewrite rule in which a node is split into two. The red (blue resp.) dotted edge indicates that there can be any number of incoming (outgoing resp.) edges (zero edges are also allowed) from nodes of the host graph to the a-looped node. Figure 3 shows the result of applying the rewrite rule to graph G of Figure 1.

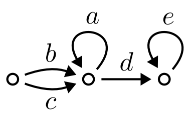

Figure 1: Graph G.

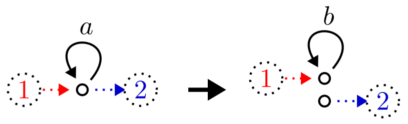

Figure 2: A PGR rule.

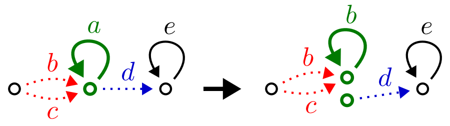

Figure 3: Applying the PGR rule to G.

Besides redirecting context, as shown in the example, it is also possible to disconnect context, direct two different nodes to the same context nodes, or change the direction of the patch edges. Context arrows are also allowed between two nodes, indicating that any number of edges can connect the two nodes or as a self-loop.

Patch graph rewriting is defined on directed, edge-labeled graphs. We can add node labels by adding labeled self-loops. In the previous example, we indicated context with a numbered dotted node. For simplicity, we will omit the dotted node in the following examples and number the dotted edges instead. Moreover, if all possible patch edges are allowed (from nodes to nodes, between nodes, and self-loops) and are not manipulated by the rule, we can omit them.

[2]The authors granted us written permission to use images from their paper "Patch Graph Rewriting" [23].

B Action Masking

Consider an agent tasked with putting a pan on the table. We want to specify (generically, though here we focus on the concrete setting) that a hot pan can never be put directly on the table and must instead be put on a trivet. Figure 4 shows a rewrite rule that removes all incoming edges of states in which a pan that is hot is put directly on the table. This modification prevents a planner using a policy or value network from considering this action, even if it has the highest value. The modification can be re-integrated into the network to avoid negatively impacting performance too much, similar to the setting with LTL safety constraints in [12].

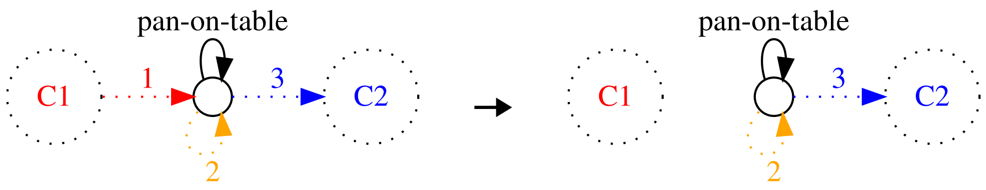

Figure 4: PGR rewrite rule that prohibits putting a pan directly on the table.

C Request Insertion

Consider a scenario with potentially harmful states. We can use PGR rules to require the agent to ask permission before executing an action that results in one of these states. Figure 5 shows an example of such a rewrite rule applied directly to the world model. Except for illustratory purposes, rewrites rules modify a controller that, when combined with the agent, would have this world model.

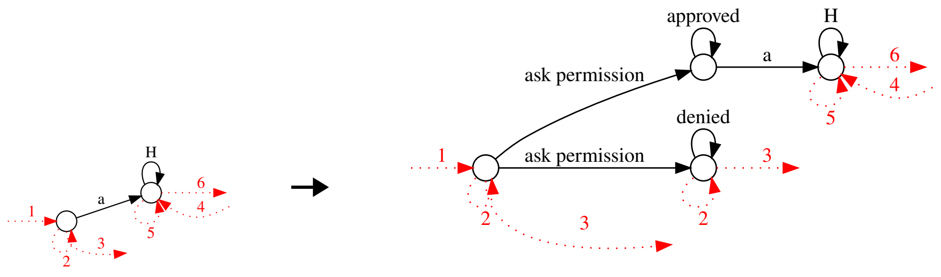

Figure 5: PGR rule that enforces human permission to execute action a. For compactness, context nodes are shown as labeled half edges.

In this instance, a represents an action that could potentially lead to a harmful (H) state. The rewrite rule introduces the requirement for the agent to seek permission before executing action a. If the request is denied, the agent is free to choose any other action available in the previous state, except for action a.

Notice how PGR rules allow us to express how newly created nodes should be connected to existing contexts. This example rewrite rule can be altered such that all actions that lead to a harmful state require permission (by making the a-labeled edge in Figure 5 a context edge) or such that permission should always be asked before executing action a, independent of whether it leads to a harmful state or not (by removing the H-loop).

In a non-deterministic environment, this paradigm can mandate the agent to ask permission whenever an action has a high probability of leading to a harmful state.

D Add Preference

Let's say we want to express that it is preferred to reach a destination state without visiting any harmful states. Therefore, we want to mark all paths that go from a starting state to a destination state that do not pass through a harmful state as "preferred". We can achieve this by using five rewrite rules.

For simplicity, we assume that harmful states are marked with a self-loop H and harmless states with a self-loop $\neg H$. We also assume that all edges in the graph are labeled O (original). Note that these markings can also be added by PGR rules that mark states as harmful based on other contexts in the transition system.

Each of these rewrite rules should be applied until a fixpoint is reached before moving on to the next rewrite rule.

0. **Pre-processing: Connected = reachable:** If an original edge connects node x and y in the transition diagram, we add an R-labeled edge to mark that y is reachable from x.
1. **Harmless transitive closure:** Iteratively connect every two nodes that have a safe path connecting the two with an R (reachability) edge. (Figure 6)
2. **Mark direction:** If there is either an original edge or a newly added R-labeled edge that connects the initial state with the goal state, we label it as RP (reachable preferred). If there is no such path, we conclude that it is impossible to reach the goal without passing a harmful state. (Figure 7)
3. **Lift preferred reachable:** Iteratively lift the RP-edges to any path that connects the start and destination state without visiting harmful states. (Figure 8)
4. **Mark preferred edges:** If there are both an original edge and an RP-labeled edge between two nodes, it is a preferred transition. (Figure 9)

Since in the rewrite rules almost all patch edges are allowed and patch edges are not reconnected in any way, we did not mark the context and patch edges in the figures for simplicity. If a patch edge is not allowed, we mark it by adding a dotted edge and crossing it.

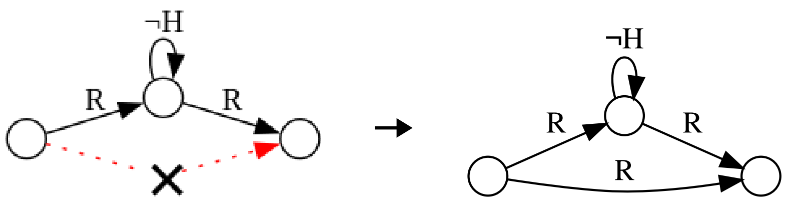

Figure 6: First rule - By iteratively applying this rule, any two nodes that have a path between them that does not visit a harmful state are connected with a reachability edge. A crossed edge indicates that no context is allowed between two states.

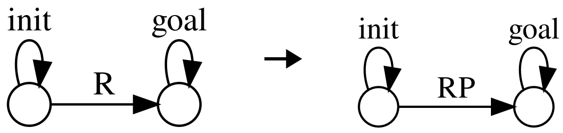

Figure 7: Second rule - Going from the start point to the goal without crossing other nodes is inherently safe and marked as "Reachable Preferred".

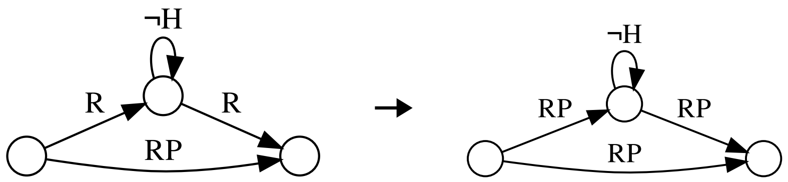

Figure 8: Third rule - Lift Reachable Preferred label to paths that connect two nodes without visiting a harmful state.

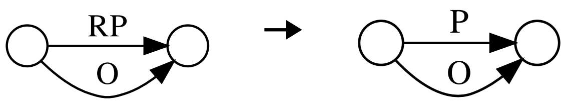

Figure 9: Fourth rule - An original edge that is Reachable Preferred is Preferred.

E Simplify Büchi Automata

E.1 LTL to Büchi Automata

A (non-deterministic) Büchi automaton is an automaton that accepts or rejects infinite words. A word is accepted if one of the paths it forms visits accepting states of the automaton infinitely often.

Any LTL formula can be translated into a Büchi automaton, a translation often relied on in LTL model checking algorithms. For an LTL formula f that uses propositions from a set PV, $\Sigma := 2^{PV \cup \neg PV}$ is the alphabet of the corresponding Büchi automaton, in which $\neg PV := \{\neg p | p \in PV\}$, the negation of all propositions in PV. In the automaton graph embedding, the state label can move through an s-labeled edge only if all propositions in s hold

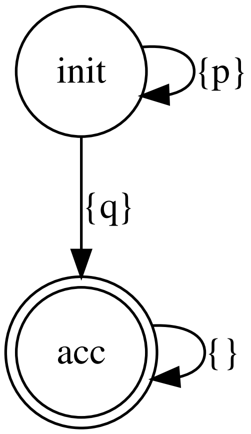

Figure 10: Büchi automaton translation of the LTL formula pUq.

in the current state.

Consider the LTL formula pUq, which states that q should hold somewhere in the future, and p has to hold until then. The formula uses the alphabet $\Sigma = 2^{\{p,q,\neg p,\neg q\}}$. In Figure 10 it is clear that to visit the accepting state infinitely often, p has to hold until at some point q gets true, after which no proposition will exit the accepting state.

E.2 Simplification

The Büchi automata constructed from LTL formulae with, e.g., Gerth et al.'s algorithm [10] are far from optimal in size. Figure 11 shows four PGR rules for simplifying Büchi automata, resulting in increased interpretability and verification speed. Although we proved our rules sound via ω-regular languages, they might not be complete yet.

We added a natural extension to PGR in which multiple LHS nodes can be connected to the same context.

For each rewrite rule, we add special cases of these rules in which one or more of the nodes are initial or accepting states. When both nodes on the LHS are initial (resp. accepting), all rules still hold if the RHS node is also initial (resp. accepting).

To illustrate, we expand rule C to accommodate the special cases. If no context is empty, and only one of the two LHS nodes is marked as accepting, the rewrite rule does not hold. However, if contexts 0 and 3 are empty, and the RHS node is accepting, the rewrite does hold. We show these special cases if rule C in Figure 12. The other rules have similar special cases.

To prevent rules from matching context self-loops with the accepting or initial state indications, we add labeled self-loops to each state of the automaton that is not initial or accepting and to the rule's nodes. Alternatively, we can use the extension of R. Overbeek and J. Endrullis [23] in which it is possible to indicate that a certain labeled edge cannot be present.

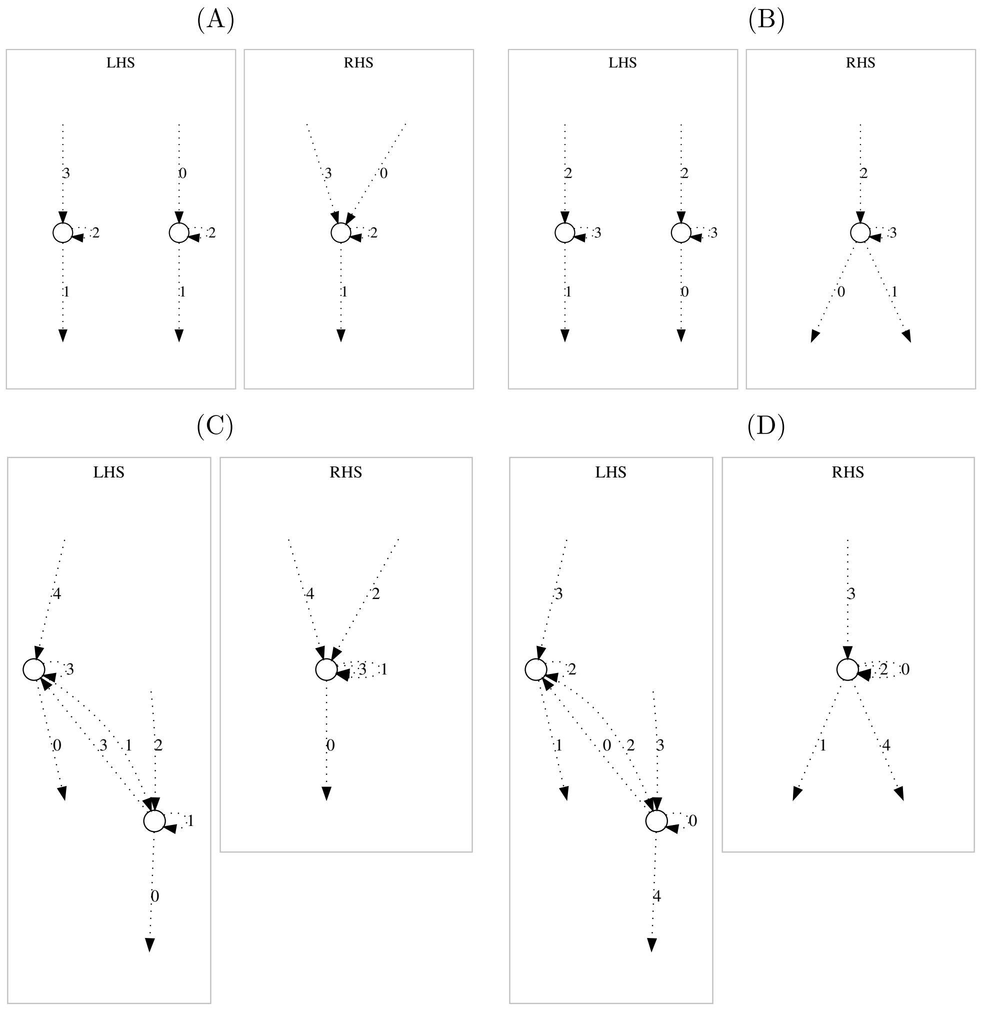

Figure 11: Four rewrite rules to simplify Büchi automata.

Consider the LTL formula $((Xp)Uq)$, stating that q has to hold somewhere in the future, and p should always hold in the next state until then. Using Gerth et al.'s algorithm [10], the constructed Büchi automaton is depicted in figure 13. Using the rewrite rule from Figure 11C and its special cases, we can simplify the Büchi automaton significantly resulting in the automaton on the right-hand side of Figure 16. Figures 14, 15 and 16 show the simplification steps.

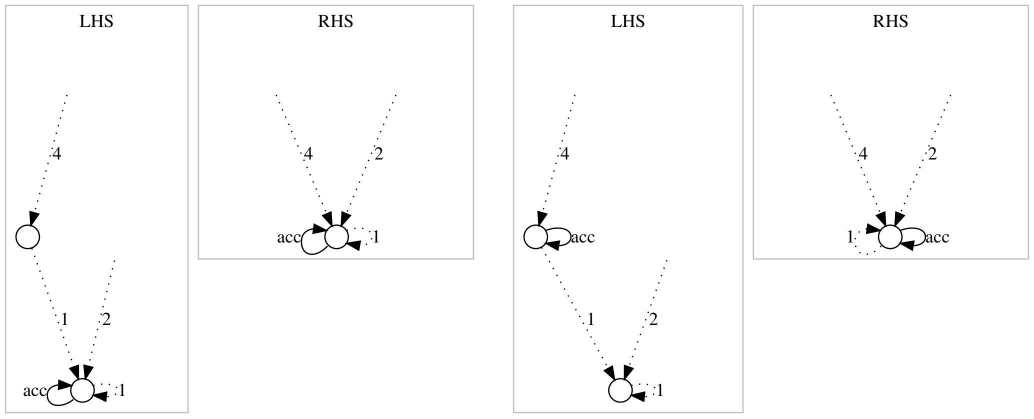

Figure 12: Two special cases of rule C.

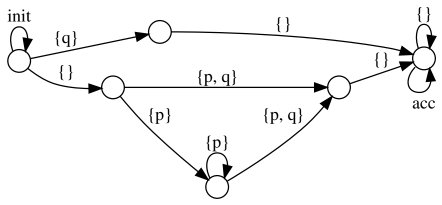

Figure 13: The Büchi automaton of the LTL formula $((Xp)Uq)$ constucted by the Gert et. al algorithm [10].

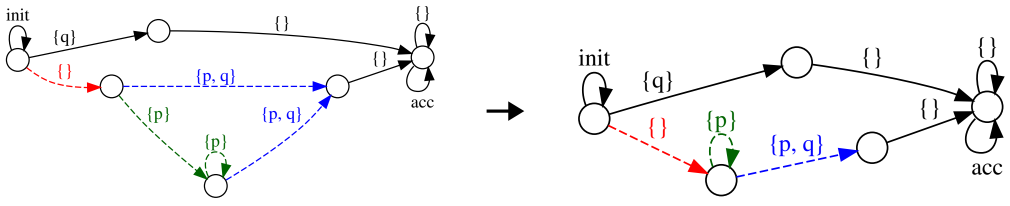

Figure 14: Applying rule C to the Büchi automaton.

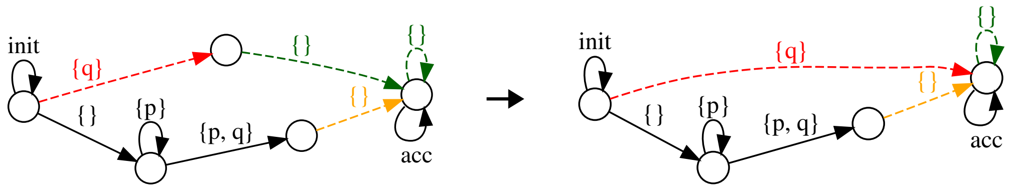

Figure 15: Applying the special case of rule C in which context 0 and 3 are empty, and one of the nodes is accepting.

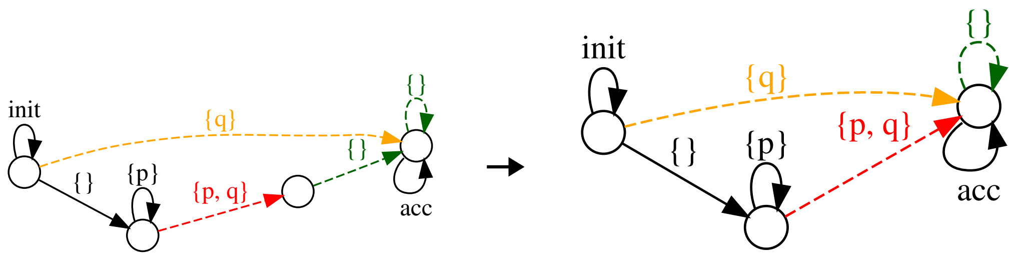

Figure 16: Applying the special case of rule C one last time to obtain a fully simplified Büchi automaton of the LTL formula $((Xp)Uq)$.

F Path Checking

Using PGR rules, it is possible to traverse two subgraphs simultaneously. This technique can be used to verify whether a finite path (be it an execution trace or a sequence of consequences of planned actions) - henceforth "trace" - satisfies an LTL$_f$ formula. Like translating an LTL formula to a Büchi automaton, an LTL$_f$ formula can be translated to a Deterministic Finite Automaton (DFA). Assume the DFAs are made deterministic and complete, simplifying the rewrite rules.

Let PV be a set of propositions, and let the trace be represented by a "null-terminated" path graph where each node is labeled with a subset of PV. The LTL$_f$ formula and corresponding DFA are defined over PV. The first node of the path is labeled with *now*, and the node corresponding to the initial state of the DFA is labeled with *state*.

Figure 17 and 18 show rewrite rules to traverse the trace and the DFA simultaneously. A path satisfies an LTL$_f$ formula if and only if no more rewrites apply and the node labeled *state* also has the *acc* (accepting) label.

We show two examples in figures 19 and 20. Applying the rewrite rules in Figure 19 will indicate that the formula $F(a \wedge b)$ holds on the path. Figure 20 shows an example of the formula $cRa := \neg(\neg cU\neg a)$, which does not hold.

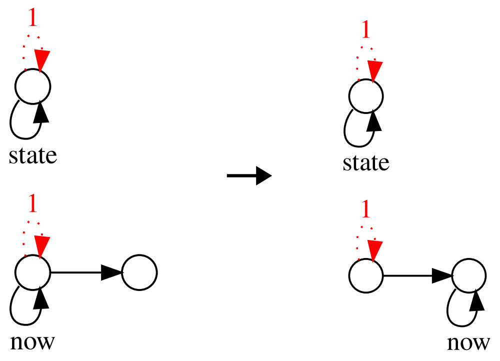

Figure 17: Rule 1 - PGR rule to traverse a trace and DFA simultaneously. Stays in the same state via the self loop holding on the trace.

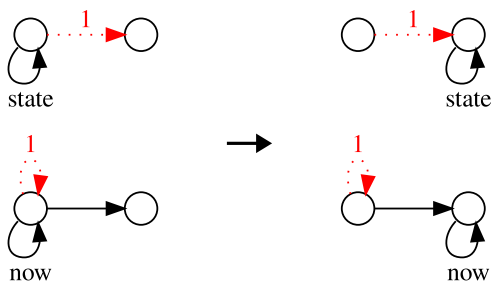

Figure 18: Rule 2 - PGR rule to traverse a trace and DFA simultaneously.

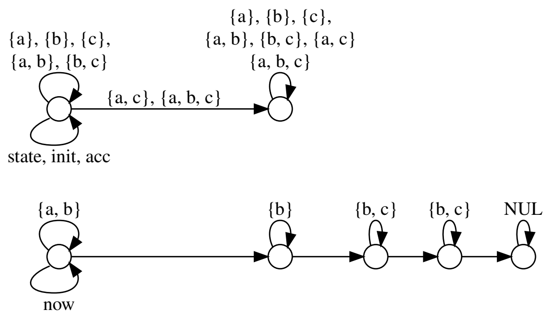

Figure 19: Automaton of the LTL formula $G(\neg(a \wedge c))$, and a trace to check it on. Comma separation of labels is used for compactness and expands to multiple edges with one label each.

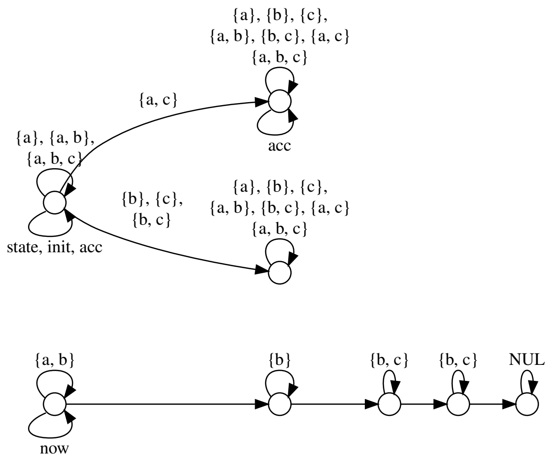

Figure 20: Automaton of the LTL formula $cRa := \neg(\neg cU\neg a)$, and a trace to check it on. Comma separation of labels is used for compactness and expands to multiple edges with one label each.