Rabinizer 2: Small Deterministic Automata for $\mathrm{LTL}_{\setminus\mathbf{GU}}$

Jan Křetínský[1,2,⋆] and Ruslán Ledesma Garza[1,⋆⋆]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. We present a tool that generates automata for LTL(**X**,**F**,**G**,**U**) where **U** does not occur in any **G**-formula (but **F** still can). The tool generates deterministic generalized Rabin automata (DGRA) significantly smaller than deterministic Rabin automata (DRA) generated by state-of-the-art tools. For complex properties such as fairness constraints, the difference is in orders of magnitude. DGRA have been recently shown to be as useful in probabilistic model checking as DRA, hence the difference in size directly translates to a speed up of the model checking procedures.

1 Introduction

Linear temporal logic (LTL) is a very useful and appropriate language for specifying properties of systems. In the verification process that follows the automata-theoretic approach, an LTL formula is first translated to an ω-automaton and then a product of the automaton and the system is constructed and analyzed. The automata used here are typically non-deterministic Büchi automata (NBA) as they recognize all ω-regular languages and thus also LTL languages. However, for two important applications, *deterministic* ω-automata are important: probabilistic model checking and synthesis of reactive modules for LTL specifications. Here deterministic Rabin automata (DRA) are typically used as deterministic Büchi automata are not as expressive as LTL. In order to transform an NBA to a DRA, one needs to employ either Safra's construction (or some other exponential construction). This approach is taken in `PRISM` [7] a leading probabilistic model checker, which reimplements the optimized Safra's construction of `ltl2dstar` [4]. However, a straight application of this very general construction often yields unnecessarily large automata and thus also large products, often too large to be analyzed.

In order to circumvent this difficulty, one can focus on fragments of LTL. The most prominent ones are GR(1)—a restricted, but useful fragment of LTL(**X**,**F**,**G**) allowing for fast synthesis—and fragments of LTL(**F**,**G**) as investigated in e.g. [1]. Recently [6], we showed how to construct DRA from LTL(**F**,**G**) directly without NBA. As we argued there, this is an interesting fragment also

⋆ The author is supported by the Czech Science Foundation, grant No. P202/12/G061.
⋆⋆ The author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

D. Van Hung and M. Ogawa (Eds.): ATVA 2013, LNCS 8172, pp. 446–451, 2013.
© Springer International Publishing Switzerland 2013

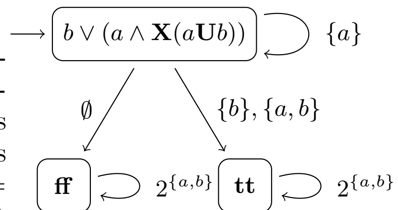
because it can express all complex fairness constraints, which are widely used in verification. We implemented our approach in a tool `Rabinizer` [3] and observed significant improvements, especially for complex formulae: for example, for a conjunction of three fairness constraints `ltl2dstar` produces a DRA with more than a milion states, while `Rabinizer` produces 469 states. Moreover, we introduced a new type of automaton a *deterministic generalized Rabin automaton* (DGRA), which is an intermediate step in our construction, and only has 64 states in the fairness example and only 1 state if transition acceptance is used. In [2], we then show that for probabilistic model checking DGRA are not more difficult to handle than DRA. Hence, without tradeoff, we can use often much smaller DGRA, which are only produced by our construction.

Here, we present a tool `Rabinizer 2` that extends our method and implements it for $\text{LTL}_{\setminus\mathbf{GU}}$ a fragment of LTL($\mathbf{X}$,$\mathbf{F}$,$\mathbf{G}$,$\mathbf{U}$) where $\mathbf{U}$ are not inside $\mathbf{G}$-formulae (but $\mathbf{F}$ still can) in negation normal form. This fragment is not only substantially more complex, but also practically more useful. Indeed, with the unrestricted $\mathbf{X}$-operator, it covers GR(1) and can capture properties describing local structure of systems and is necessary for description of precise sequences of steps. Further, $\mathbf{U}$-operator allows to distinguish paths depending on their initial parts and then we can require different fairness constraints on different paths such as in $wait\mathbf{U}(answer_1 \wedge \phi_1) \vee wait\mathbf{U}(answer_2 \wedge \phi_2)$ where ϕ_1, ϕ_2 are two fairness constraints. As another example, consider patterns for "before": for "absence" we have $\mathbf{F}r \rightarrow (\neg p\mathbf{U}r)$, for "constrained chains" $\mathbf{F}r \rightarrow (p \rightarrow (\neg r\mathbf{U}(s \wedge \neg r \wedge \neg z \wedge \mathbf{X}((\neg r \wedge \neg z)\mathbf{U}t))))\mathbf{U}r$.

Furthermore, as opposed to other tools (including `Rabinizer`), `Rabinizer 2` can also produce DGRA, which are smaller by orders of magnitude for complex formulae. For instance, for a conjunction of four fairness constraints the constructed DGRA has 256 states, while the directly degeneralized DRA is 20736-*times* bigger [2]. As a result, we not only obtain smaller DRA now for much larger fragment (by degeneralizing the DGRA into DRA), but also the power of DGRA is made available for this fragment allowing for the respective speed up of probabilistic model checking.

The tool can be downloaded and additional materials and proofs found at `http://www.model.in.tum.de/~kretinsk/rabinizer2.html`

2 Algorithm

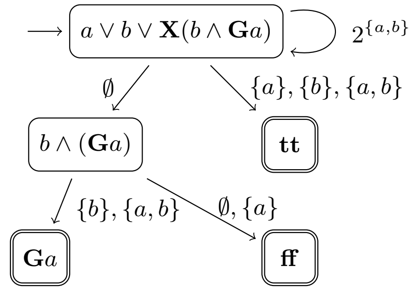


Let us fix a formula φ of $\text{LTL}_{\setminus\mathbf{GU}}$. We construct an automaton $\mathcal{A}(\varphi)$ recognizing models of φ. Details can be found on the tool's webpage. In every step, $\mathcal{A}(\varphi)$ unfolds φ as in [6], now we also define $\mathfrak{Unf}(\psi_1\mathbf{U}\psi_2) = \mathfrak{Unf}(\psi_2) \vee (\mathfrak{Unf}(\psi_1) \wedge \mathbf{X}(\psi_1\mathbf{U}\psi_2))$. Then it checks whether the letter currently read complies with thus generated requirements, see the example on the right for $\varphi = a\mathbf{U}b$. E.g. reading $\{a\}$ yields requirement $\mathbf{X}(a\mathbf{U}b)$ for the next step, thus in the next step we have $\mathfrak{Unf}(a\mathbf{U}b)$ which is the same as in the initial state, hence we loop.

Some requirements can be checked at a finite time by this unfolding, such as $b\mathbf{U}(a \wedge \mathbf{X}b)$, some cannot, such as $\mathbf{GF}(a \wedge \mathbf{X}b)$. The state space has to monitor the latter requirements (such as the repetitive satisfaction of $a \wedge \mathbf{X}b$) separately. To this end, let $\mathbf{G}_\varphi := \{\mathbf{G}\psi \in \mathrm{sf}(\varphi)\}$ and $\mathbf{F}_\varphi := \{\mathbf{F}\psi \in \mathrm{sf}(\omega) \mid \text{for some } \omega \in \mathbf{G}_\varphi\}$ where $\mathrm{sf}(\varphi)$ denotes the set of all subformulae of φ. Then $\mathcal{R}\mathrm{ec} := \{\psi \mid \mathbf{G}\psi \in \mathbf{G}_\varphi \text{ or } \mathbf{F}\psi \in \mathbf{F}_\varphi\}$ is the set of *recurrent* subformulae of φ, whose repeated satisfaction we must check. (Note that no $\mathbf{U}$ occurs in formulae of $\mathcal{R}\mathrm{ec}$.) In the case without the $\mathbf{X}$ operator [6,3], such as with $\mathbf{GF}a$, it was sufficient to record the currently read letter in the states of $\mathcal{A}(\varphi)$. Then the acceptance condition checks whether e.g. a is visited infinitely often. Now we could extend this to keep history of the last n letters read where n is the nesting depth of the $\mathbf{X}$ operator in φ. In order to reduce the size of the state space, we rather store equivalence classes thereof. This is realized by automata. For every $\xi \in \mathcal{R}\mathrm{ec}$, we have a finite automaton $\mathcal{B}(\xi)$, and $\mathcal{A}(\varphi)$ will keep track of its current states.

Construction of $\mathcal{B}(\xi)$: We define a finite automaton $\mathcal{B}(\xi) = (Q_\xi, i_\xi, \delta_\xi, F_\xi)$ over 2^{Ap} by

- the set of states $Q_\xi = \mathsf{B}^+(\mathrm{sf}(\xi))$, where $\mathsf{B}^+(S)$ is the set of positive Boolean functions over S and $\mathbf{tt}$ and $\mathbf{ff}$,
- the initial state $i_\xi = \xi$,
- the final states F_ξ where each atomic proposition has $\mathbf{F}$ or $\mathbf{G}$ as an ancestor in the syntactic tree (i.e. no atomic propositions are guarded by only $\mathbf{X}$'s and Boolean connectives),
- transition relation δ_ξ is defined by transitions



$$\chi \xrightarrow{\nu} \mathbf{X}^{-1}(\chi[\nu]) \quad \text{for every } \nu \subseteq Ap \text{ and } \chi \notin F$$
$$i \xrightarrow{\nu} i \quad \text{for every } \nu \subseteq Ap$$

where $\chi[\nu]$ is the function χ with $\mathbf{tt}$ and $\mathbf{ff}$ plugged in for atomic propositions according to ν and $\mathbf{X}^{-1}\chi$ strips away the initial $\mathbf{X}$ (whenever there is one) from each formula in the Boolean combination χ. Note that we do not unfold inner $\mathbf{F}$- and $\mathbf{G}$-formulae. See an example for $\xi = a \vee b \vee \mathbf{X}(b \wedge \mathbf{G}a)$ on the right.

Construction of $\mathcal{A}(\varphi)$: The state space has two components. Beside the component keeping track of the input formula, we also keep track of the history for every recurrent formula of $\mathcal{R}\mathrm{ec}$. The second component is then a vector of length $|\mathcal{R}\mathrm{ec}|$ keeping the current set of states of each $\mathcal{B}(\xi)$. Formally, we define $\mathcal{A}(\varphi) = (Q, i, \delta)$ to be a deterministic finite automaton over $\Sigma = 2^{Ap}$ given by

- set of states $Q = \mathsf{B}^+(\mathrm{sf}(\varphi) \cup \mathbf{X}\mathrm{sf}(\varphi)) \times \prod_{\xi \in \mathcal{R}\mathrm{ec}} 2^{Q_\xi}$ where $\mathbf{X}S = \{\mathbf{X}s \mid s \in S\}$,
- the initial state $i = \langle \mathfrak{Unf}(\varphi), (\xi \mapsto \{i_\xi\})_{\xi \in \mathcal{R}\mathrm{ec}} \rangle$;
- the transition function δ is defined by transitions

$$\langle \psi, (R_\xi)_{\xi \in \mathcal{R}\mathrm{ec}} \rangle \xrightarrow{\nu} \langle \mathfrak{Unf}(\mathbf{X}^{-1}(\psi[\nu])), \big(\delta_\xi(R_\xi, \nu)\big)_{\xi \in \mathcal{R}\mathrm{ec}} \rangle$$

On $\mathcal{A}(\varphi)$ it is possible to define an acceptance condition such that $\mathcal{A}(\varphi)$ recognizes models of φ. The approach is similar to [6], but now we have to take the information of each $\mathcal{B}(\xi)$ into account. We use this information to get look-ahead necessary for evaluating $\mathbf{X}$-requirements in the first component of $\mathcal{A}(\varphi)$. However, since storing complete future look-ahead would be costly, $\mathcal{B}(\xi)$ actually stores the compressed information of past. The acceptance condition allows then for deducing enough information about the future.

Further optimizations include not storing states of each $\mathcal{B}(\xi)$, but only the currently relevant ones. E.g. after reading $\emptyset$ in $\mathbf{GF}a \vee (b \wedge \mathbf{GF}c)$, it is no more interesting to track if c occurs infinitely often. Further, since only the infinite behaviour of $\mathcal{B}(\xi)$ is important and it has acyclic structure (except for the initial states), instead of the initial state we can start in any subset of states. Therefore, we start in a subset that will occur repetitively and we thus omit unnecessary initial transient parts of $\mathcal{A}(\varphi)$.

3 Experimental Results

We compare our tool to `ltl2dstar`, which yields the same automata as its Java reimplementation in PRISM. We consider some formulae on which `ltl2dstar` was originally tested [5], some formulae used in a network monitoring project Liberouter (https://www.liberouter.org/) showing the $\text{LTL}_{\setminus\mathbf{GU}}$ fragment is practically very relevant, and several other formulae with more involved structure such as ones containing fairness constraints. For results on the LTL($\mathbf{F}$,$\mathbf{G}$) subfragment, we refer to [3]. Due to [2], it only makes sense to use DGRA and we thus display the sizes of DGRA for `Rabinizer 2` (except for the more complex cases this, however, coincides with the degeneralized DRA). Here "?" denotes time-out after 30 minutes. For more experiments, see the webpage.

Formula	`ltl2d*`	`R.2`
$(\mathbf{F}p)\mathbf{U}(\mathbf{G}q)$	4	3
$(\mathbf{G}p)\mathbf{U}q$	5	5
$\neg(p\mathbf{U}q)$	4	3
$\mathbf{G}(p \to \mathbf{F}q) \wedge ((\mathbf{X}p)\mathbf{U}q) \vee \neg\mathbf{X}(p\mathbf{U}(p \wedge q))$	19	8
$\mathbf{G}(q \vee \mathbf{XG}p) \wedge \mathbf{G}(r \vee \mathbf{XG}\neg p)$	5	14
$((\mathbf{G}(\mathbf{F}(p_1) \wedge \mathbf{F}(\neg p_1)))) \to (\mathbf{G}((p_2 \wedge \mathbf{X}p_2 \wedge \neg p_1 \wedge \mathbf{X}p_1 \to ((p_3) \to \mathbf{X}p_4))))$	11	8
$((p_1 \wedge \mathbf{XG}(\neg p_1)) \wedge (\mathbf{G}((\mathbf{F}p_2) \wedge (\mathbf{F}\neg p_2))) \wedge ((\neg p_2))) \to (((\neg p_2)\mathbf{U}$ $\mathbf{G}(\neg((p_3 \wedge p_4) \vee (p_3 \wedge p_5) \vee (p_3 \wedge p_6) \vee (p_4 \wedge p_5) \vee (p_4 \wedge p_6) \vee (p_5 \wedge p_6)))))$	17	8
$(\mathbf{X}p_1 \wedge \mathbf{G}((\neg p_1 \wedge \mathbf{X}p_1) \to XXp_1) \wedge \mathbf{GF}\neg p_1 \wedge \mathbf{GF}p_2 \wedge \mathbf{GF}\neg p_2) \to$ $(\mathbf{G}(p_3 \wedge p_4 \wedge !p_2 \wedge \mathbf{X}p_2 \to \mathbf{X}(p_1 \vee \mathbf{X}(\neg p_4 \vee p_1))))$	9	7
$\mathbf{F}r \to (p \to (\neg r\mathbf{U}(s \wedge \neg r \wedge \neg z \wedge \mathbf{X}((\neg r \wedge \neg z)\mathbf{U}t))))\mathbf{U}r$	6	5
$((\mathbf{GF}(a \wedge \mathbf{XX}b) \vee \mathbf{FG}b) \wedge \mathbf{FG}(c \vee (\mathbf{X}a \wedge \mathbf{XX}b)))$	353	73
$\mathbf{GF}(\mathbf{XXX}a \wedge \mathbf{XXXX}b) \wedge \mathbf{GF}(b \vee \mathbf{X}c) \wedge \mathbf{GF}(c \wedge \mathbf{XX}a)$	2127	85
$(\mathbf{GF}a \vee \mathbf{FG}b) \wedge (\mathbf{GF}c \vee \mathbf{FG}(d \vee \mathbf{X}e))$	18176	40
$(\mathbf{GF}(a \wedge \mathbf{XX}c) \vee \mathbf{FG}b) \wedge (\mathbf{GF}c \vee \mathbf{FG}(d \vee \mathbf{X}a \wedge \mathbf{XX}b))$	?	142
$a\mathbf{U}b \wedge (\mathbf{GF}a \vee \mathbf{FG}b) \wedge (\mathbf{GF}c \vee \mathbf{FG}d) \vee a\mathbf{U}c \wedge (\mathbf{GF}a \vee \mathbf{FG}d) \wedge (\mathbf{GF}c \vee \mathbf{FG}b)$	?	60

References

1. Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. ACM Trans. Comput. Log. 5(1), 1–25 (2004)
2. Chatterjee, K., Gaiser, A., Křetínský, J.: Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 559–575. Springer, Heidelberg (2013)
3. Gaiser, A., Křetínský, J., Esparza, J.: Rabinizer: Small deterministic automata for lTL(F,G). In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 72–76. Springer, Heidelberg (2012)
4. Klein, J.: ltl2dstar - LTL to deterministic Streett and Rabin automata, `http://www.ltl2dstar.de/`
5. Klein, J., Baier, C.: Experiments with deterministic *omega*-automata for formulas of linear temporal logic. Theor. Comput. Sci. 363(2), 182–195 (2006)
6. Křetínský, J., Esparza, J.: Deterministic automata for the (F,G)-fragment of LTL. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 7–22. Springer, Heidelberg (2012)
7. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)