`ltlsynt` (Spot 2.9+)

Florian Renkin
LRDE, EPITA
Le Kremlin-Bicêtre, France
frenkin@lrde.epita.fr

Alexandre Duret-Lutz
LRDE, EPITA
Le Kremlin-Bicêtre, France
adl@lrde.epita.fr

Adrien Pommellet
LRDE, EPITA
Le Kremlin-Bicêtre, France
adrien@lrde.epita.fr

Philipp Schlehuber
LRDE, EPITA
Le Kremlin-Bicêtre, France
philipp@lrde.epita.fr

1 Introduction and History

The tool `ltlsynt` distributed in the Spot library[6] since version 2.5 was originally developed by Thibaud Michaud and Maximilien Colange. They submitted it to the 2017 and 2018 [9] editions of the SYNTCOMP. This short document summarizes the improvements brought to `ltlsynt` since then.

While both original authors left the project mid-2018 (the former graduated and the latter moved to the industry), Maximilien had started working on an alternative approach called LAR (described below) that was eventually included in the Spot 2.7 release. Without any submission of `ltlsynt` to SYNTCOMP'19, the organizers installed the latest version distributed with Spot 2.7.4 themselves, and uncovered a bug caused by an incorrect optimization in the LAR approach.

This optimization was simply reverted in Spot 2.8, and we started working on a reimplementation of LAR with many optimizations for Spot 2.9. Moreover, this submission contains additional improvements that have not been released yet and should be part of a future release, hence the "2.9+" name.

A quick summary of all versions submitted to SYNTCOMP over the years is given in Table 1.

2 Technical Details

We describe `ltlsynt`'s general approach in Figure 1. The main step of the synthesis process is to convert the LTL specification constraining the input and output signals into a deterministic parity automaton (DPA) where transitions labeled by Boolean combinations of input signals are followed by transitions labeled by Boolean combinations of output signals, as shown in the blue-colored box and discussed in Section 2.1. This DPA uses a transition-based max-odd parity acceptance, i.e., assuming its transitions are also labeled by priorities (a.k.a. colors), only runs whose highest priority is odd are accepting. We then interpret this DPA as a game between two players (the *environment* playing the input signals and the *controller* playing the output signals) and search a winning strategy for the *controller* using a transition-based version of Zielonka's algorithm [11], then encode this strategy as an AIGER circuit.

Year	Spot version	Main news
2017	pre-2.4? + patches	first implementation
2018	2.5.3 + patches	optimizations to determ., and game solving; incr. determ. approach
2019	2.7.4	(bogus) LAR; improved LTL translation; incr. determ. removed
2020	2.9 + patches	reimplemented LAR, split, and game solving; parity minimization

Table 1: Versions of Spot on which `ltlsynt` submissions to SYNTCOMP where based.

Submitted to:
Syntcomp 2020

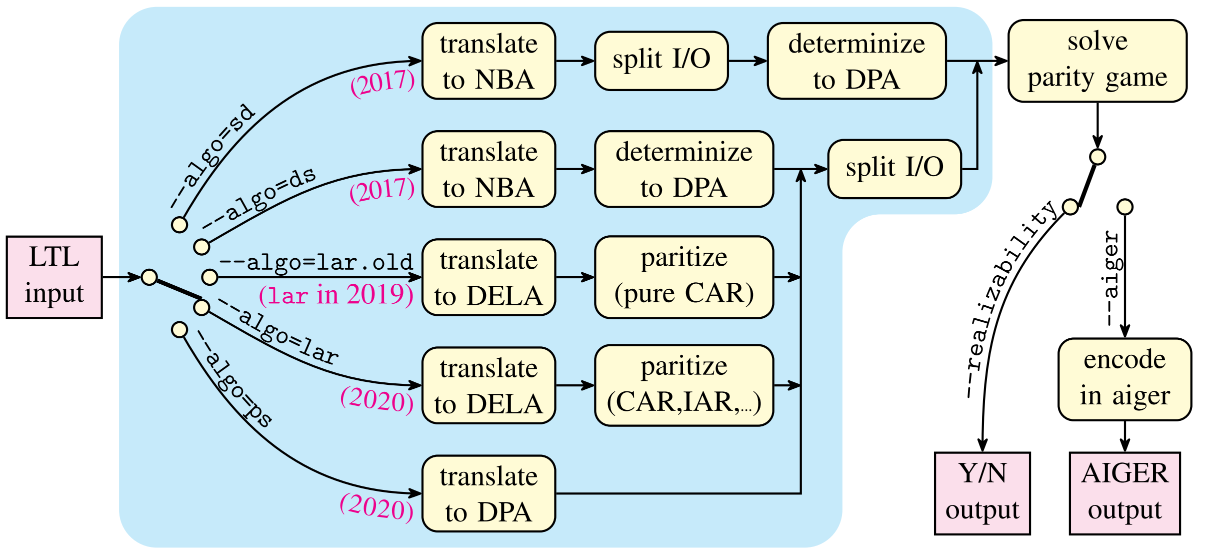

Figure 1: The `--algo` option of `ltlsynt` selects between different pipelines for building a DPA.

2.1 Determinization pipelines

The algorithm `ltlsynt` uses to convert the LTL input into a DPA suitable for game solving depends on the `--algo` command-line argument. The first two options, `ds` and `sd`, correspond to pipelines that appeared in `ltlsynt`'s very first release. With `--algo=ds`, LTL inputs are first converted to non-deterministic Büchi automata, then determinized to DPA using a variant of Safra. At this point, transitions are labeled by an mix of input and output signals, so transitions of the form $\bigcirc \xrightarrow{i_1\land i_2\land o_1\land o_2} \bigcirc$ are split into $\bigcirc \xrightarrow{i_1\land i_2} \bigcirc \xrightarrow{o_1\land o_2} \bigcirc$. To preserve determinism, we ensure that multiple transitions sharing the same inputs end up sharing the same new intermediate state. In the pipeline `--algo=sd`, we perform this split before determinizing the automata. Intuitively, this choice may be explained by realizing that the determinization function, in order to compute the successors of a given state, has to consider all compatible assignments of the atomic propositions used by transitions leaving said state: in `ds`, there might be up to $2^{|I|+|O|}$ assignments to consider, whereas in `sd` a given state has at most either $2^{|I|}$ or $2^{|O|}$ possible successors.

The option `--algo=lar.old` in Spot 2.9 was called `--algo=lar` in Spot 2.7 and 2.8, and relies on Spot's ability to translate LTL formulas into automata with Emerson-Lei acceptance condition (i.e., any acceptance condition). To do so, this algorithm decomposes the input LTL formula on Boolean operators, translates sub-formulas into deterministic automata (possibly using algorithms specialized for a particular class of formulas), recombines the resulting automata using synchronous products, then applies the relevant Boolean operations on the acceptance conditions. If we are lucky enough, we may avoid Safra-based determinization entirely. However, the resulting deterministic automaton may feature some arbitrary conditions that have yet to be paritized. Therefore, we use a transition-based adaptation of the *state appearance record* algorithm, typically used to convert state-based Muller acceptance to state-based parity. This option was named LAR as a reference to the *latest appearance record* family of algorithms to which SAR belongs (to the extent some variants of SAR are often called LAR). In a paper submitted to ATVA'20, we have renamed this algorithm CAR, because our method actually uses a *color appearance record*, i.e., it tracks only the colors but not the states nor the transitions.

The CAR implementation in Spot 2.7 featured an optimization that reduced the number of colors tracked by computing the classes of symmetric colors in the acceptance condition (two colors are symmetric if swapping them in the acceptance formula results in an equivalent formula). The intent was to keep track of a smaller number of acceptance classes instead of colors, but this optimization was found to be incorrect during SYNTCOMP'19. This optimization was removed from Spot 2.8 for correctness sake, then replaced by many new optimizations in Spot 2.9.

In this version of `ltlsynt`, option `--algo=lar` triggers a new implementation of the paritization procedure (described in our ATVA'20 submission). It combines CAR (a generic transformation to parity) with IAR (a conversion of Rabin-like or Streett-like acceptance conditions to parity) as well as a partial-degeneralization (in order to reduce conjunctions of Inf or disjunctions of Fin that occur in the acceptance condition to a single term, as intended by the original symmetry-based optimization) and multiple simplifications of the acceptance conditions. All these transformations are performed on each SCC separately, and it may for instance happen that one SCC is paritized using CAR while another SCC is partially degeneralized to produce an acceptance condition that can be paritized with IAR. Our benchmarks performed on data from SYNTCOMP'17 suggest that the option `--algo=lar` often produces significantly smaller DPAs than `--algo=lar.old`.

Finally, a new option, not yet available in Spot 2.9, is `--algo=ps`. This is a close variant of `--algo=ds`, that relies on the translation code that powers `ltl2tgba -P -D`. This procedure splits the top-level LTL formulas on Boolean operators in order to translate subformulas corresponding to obligations formulas separately. The remaining subformulas are separately translated to NBA, determinized using Safra if needed, then combined back with the obligation part. Our preliminary experiments showed this option to be inferior to the other methods, and since we had to pick three configurations for this year's competition, we excluded this procedure. In the future it could be improved by tagging the subformulas based on their corresponding acceptance conditions, as performed by Strix [8].

2.2 Various optimizations

We now discuss other optimizations that were introduced since the 2018 release.

Translation Since Spot 2.7, the LTL translation engine (which stands behind the "translate to *xx*A" boxes in Figure 1) learned to split the input formula on Boolean operators in order to separately translate parts to automata then combine these to produce the desired result. This is similar to the process used by the `delag` tool [10], but we use slightly improved algorithms. Extracting *obligations* subformulas is beneficial because those can be converted to minimal weak deterministic automata [4]. Subformulas of the form $\mathsf{GF}(\textit{guarantee})$ or $\mathsf{FG}(\textit{safety})$ can be converted to DBA or DCA using dedicated algorithms (our implementation is a crossover between two different works [7, 10]). Finally, the products combining the resulting automata handle weak-automata and suspendable properties [1] specifically. The heuristics used depend on the type of automata to produce. For instance, in order to generate NBA or DBA, we only split the LTL formula on conjunctions. The post-Spot-2.9 version submitted to SYNTCOMP also deals with xor and equivalence operators while converting to DELA (following Strix's [8] footsteps). Additionally, several new LTL simplifications rules have been added to improve the translation process.

Parity minimization Spot 2.8 features a function that minimizes the number of colors in a DPA [3], now called in `ltlsynt` once a DPA is produced, before merging states with identical successors.

Split — from automata to arenas The split operation described above transforms an automaton into a two-player arena. Even though this step is merely a technicality, benchmarks on Spot 2.9 have shown that it can consume up to 20% of the total run time. In the submitted version, this process has been optimized thanks to caching operations, as labels are often shared among multiple transitions. Moreover, the number of edges and states has been reduced by sharing the introduced intermediate states.

Solving the game The parity game solver of `ltlsynt` is derived from Zielonka's algorithm [11]. The implementation in the submitted version is inspired by the one described in [5] but adapted to transition-based acceptance. In particular, it supports (non-recursive) SCC decomposition, parity compression (also called priority compression) and detection of sub-arenas having a single parity. In the majority of the SYNTCOMP benchmarks, solving the parity game is not the bottleneck of `ltlsynt`, but nonetheless remains a crucial step as it also determines the strategy which directly influences the size of the resulting AIGER circuit.

Optimizing the output circuit For the synthesis track submission, the AIGER output of `ltlsynt` is run through `abc` for simplification [2]. This is done in the driver script for `starexec`, not by `ltlsynt` itself.

References

[1] Tomáš Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský & Jan Strejček (2013): *Compositional Approach to Suspension and Other Improvements to LTL Translation*. In: *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)*, *Lecture Notes in Computer Science* 7976, Springer, pp. 81–98, .

[2] Robert Brayton & Alan Mishchenko (2010): *ABC: An Academic Industrial-Strength Verification Tool*. In: *Proceedings of the 22nd Conference on Computer Aided Verification (CAV'10)*, Springer, pp. 24–40, .

[3] Olivier Carton & Ramón Maceiras (1999): *Computing the Rabin index of a parity automaton*. *Informatique théorique et applications* 33(6), pp. 495–505. Available at `http://www.numdam.org/item/ITA_1999__33_6_495_0/`.

[4] Christian Dax, Jochen Eisinger & Felix Klaedtke (2007): *Mechanizing the Powerset Construction for Restricted Classes of ω-Automata*. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino & Yoshio Okamura, editors: *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, *Lecture Notes in Computer Science* 4762, Springer.

[5] Tom van Dijk (2018): *Oink: An implementation and evaluation of modern parity game solvers*. In: *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*, Springer, pp. 291–308, .

[6] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault & Laurent Xu (2016): *Spot 2.0 — a framework for LTL and ω-automata manipulation*. In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, *Lecture Notes in Computer Science* 9938, Springer, pp. 122–129, .

[7] Javier Esparza, Jan Křetínský & Salomon Sickert (2018): *One Theorem to Rule Them All: A Unified Translation of LTL into ω-Automata*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*, ACM, pp. 384–393, .

[8] Michael Luttenberger, Philipp J. Meyer & Salomon Sickert (2020): *Practical Synthesis of Reactive Systems from LTL Specifications via Parity Games*. *Acta Informatica* 57, pp. 3—-36. Originally published on 21 November 2019.

[9] Thibaud Michaud & Maximilien Colange (2018): *Reactive Synthesis from LTL Specification with Spot*. In: *Proceedings of the 7th Workshop on Synthesis (SYNT'18)*. Available at `http://www.lrde.epita.fr/dload/papers/michaud.18.synt.pdf`.

[10] David Müller & Salomon Sickert (2017): *LTL to Deterministic Emerson-Lei Automata*. In Patricia Bouyer, Andrea Orlandini & Pierluigi San Pietro, editors: *Proceedings of the Eighth International Symposium on Games, Automata, Logics and Formal Verification (GandALF'17)*, *EPTCS* 256, pp. 180–194, .

[11] Wieslaw Zielonka (1998): *Infinite games on finitely coloured graphs with applications to automata on infinite trees*. *Theoretical Computer Science* 200(1–2), pp. 135–183, .