Reactive Synthesis from LTL Specification with Spot

Thibaud Michaud and Maximilien Colange
LRDE – EPITA, Le Kremlin-Bicêtre, France

tmichaud@lrde.epita.fr, colange@lrde.epita.fr

We present `ltlsynt`, a new tool for reactive synthesis from LTL specifications. It relies on the efficiency of Spot [8] to translate the input LTL specification to a deterministic parity automaton. The latter yields a parity game, which we solve with Zielonka's recursive algorithm [32].

The approach taken in `ltlsynt` was widely believed to be impractical, due to the double-exponential size of the parity game, and to the open status of the complexity of parity games resolution. `ltlsynt` ranked second of its track in the 2017 edition of the SYNTCOMP competition [17]. This demonstrates the practical applicability of the parity game approach, when backed by efficient manipulations of ω-automata such as the ones provided by Spot. We present our approach and report on our experimental evaluation of `ltlsynt` to better understand its strengths and weaknesses.

1 Introduction

LTL Reactive synthesis is the problem of finding a *controller* that reacts to the actions of an *environment* in order to always enforce a given linear time temporal (LTL) specification. A typical example is a monitoring system for a power plant: the environment actions model events triggered by the monitored parameters (such as pressure reaching a critical value) while controller actions represent the possible responses of the monitor (such as opening a safety valve). An instance of the problem is described by a set of uncontrollable actions $\mathscr{I}$, a set of controllable actions $\mathscr{O}$ and an LTL formula ϕ over the set of atomic propositions $\mathscr{I} \cup \mathscr{O}$. A solution to an instance is a function describing which controllable actions to enact, in response to arbitrary uncontrollable actions, so that the infinite sequence of actions always satisfies the specification ϕ.

This problem is known to be 2EXPTIME-complete [27], and several approaches have been proposed to solve it (see Section 2). Despite its theoretical complexity, its potential applications motivate the implementation of tools, and it appears that many practical instances could be solved efficiently. The SYNTCOMP competition [18], held every year since 2014, aims to stimulate the development of practical solutions to this problem by confronting tools on a set of benchmarks.

Spot [8] is a C++ library for the manipulation of LTL formulas and ω-automata. It features a collection of algorithms for efficient manipulation of LTL formulas and ω-automata: formula simplication, translation to ω-automata, determinization of ω-automata, simplification of ω-automata *etc*. Although primarily aimed at model-checking, all these features are of great interest to solve the LTL reactive synthesis problem. Spot has been under active development for over a decade now, and combines high-quality research results to a masterful implementation.

We describe here how we extended Spot into a reactive synthesis solver called `ltlsynt`. It uses the existing features of Spot to translate the input LTL synthesis instance to a deterministic parity automaton, interpreted as a parity game. This game, which is equivalent to the input problem, is then solved using the state-of-the-art recursive algorithm by Zielonka [32]. We hope to demonstrate that the continued effort behind Spot also has fruitful applications in synthesis.

We first review related work in Section 2. We formally define in Section 3 the reactive synthesis problem, as well as the notion of ω-automata and of parity games that will be useful in the sequel. Section 4 describes the approach implemented in `ltlsynt`, that transforms the input reactive synthesis problem to an equivalent parity game. A brief overview of the architecture and usage of `ltlsynt` is given in Section 5. We conclude by an experimental assessment of `ltlsynt` in Section 6.

2 Related Work

Synthesis, *i.e.* the automated generation of a program from its specification, can be seen as the ultimate stage of declarative programming. *LTL reactive synthesis* generates a *controller*, usually as a digital circuit, from a specification given as a formula of Linear Temporal Logic (LTL).

Two main approaches to solve LTL reactive synthesis can be identified: one by reduction to parity games, and the other by reduction to a bounded safety game. We shortly review those approaches here. `ltlsynt` uses the former one, and it will be described in more details throughout the paper.

Reactive synthesis is naturally described as a turn-based game between two players: the controller and the environment. A move for a player is a choice of signals to enact. A play is a (finite or infinite) sequence of alternated moves between both players. A play is won by the controller if it satisfies the specification, and a controller can be synthesized if the controller has a winning strategy in this game.

LTL (or more generally ω-regular) specifications can be described by finite automata on infinite words (or ω-automata). Such automata are good candidates to build finite arena for the above game, a first step towards their resolution. However, determinism is a crucial property for the automaton game to faithfully mimic the synthesis game (this fact will be detailed in Section 4). The historical approach to LTL (and ω-regular) reactive synthesis consists in building a deterministic ω-automaton from the specification and to turn it into a turn-based game with an ω-regular winning condition.

2.1 Parity games

Among all possible ω-regular winning conditions for games, the parity condition has drawn a lot of attention. Parity games are *determined*: there is always a winning strategy for one of the two players [25]. Moreover, if a Player has a winning strategy, he/she has a *memoryless* winning strategy [10]. The latter property shows that the problem of solving parity games lies in **NP**. Determinacy expresses the symmetry between the players, demonstrating that the problem also lies in co-**NP**. Whether it can be solved in polynomial time is a long-standing open problem. This is in contrast with other acceptance conditions: solving Rabin (resp. Streett) games is **NP**-complete (resp. co-**NP**-complete) [11], whereas winning strategies in Muller games require exponential memory [9].

The unknown complexity status of parity game solving, together with the widely accepted belief that it is solvable in polynomial time, has motivated a lot of research on the subject. As a result, many algorithms have been proposed [20, 22, 4, 30, 26] together with variants and optimizations, with various worst-case complexity and different favorable and unfavorable instances. It has also motivated implementation efforts: PGSolver [14] and Oink [6] are two platforms both featuring a large collection of algorithms to solve parity games. This activity recently culminated with the publication of the first quasi-polynomial time algorithm for the problem [3]. Despite its recency, this breakthrough has already inspired several works which improve its worst-case complexity [21, 15, 13].

These recent algorithmic progresses convinced us to prioritize this approach in `ltlsynt`. They also promise renewed interest and activity in the following years, and thus many leads for future improvements of our tool.

2.2 Avoiding determinization

The main drawback in the approach detailed above is the exponential blowup in size induced by the determinization of the ω-automaton. Determinization of a Büchi automaton with n states yields a deterministic parity automaton with $\mathcal{O}((n!)^2)$ states and $2n$ priorities [31]. This is the best known upper-bound to date. Solutions avoiding determinization (and this blowup) have thus been proposed.

A first proposal originates in the remark that determinism is actually too strong a condition for the built parity game to be equivalent to the original reactive synthesis instance. A weaker condition, known as the arena being *good-for-games* [16] or *history-deterministic* [5], is actually sufficient. It is known that good-for-games ω-automata can be much smaller than deterministic ones [23]. Nevertheless, this condition remains of little practical interest, as there is no known algorithm to turn an ω-automaton into a smaller-than-deterministic good-for-games one. Furthermore, the complexity of testing good-for-gameness is largely unknown: the lower bound is polynomial (for co-Büchi automata [23]) while the upper bound is exponential [16].

Another proposal consists in building a safety game to solve the reactive synthesis problem [24]. The reactive synthesis specification is first turned into a *universal co-Büchi automaton*. A run of a co-Büchi automaton is accepting if it does *not* visit the co-Büchi condition infinitely often. In a universal automaton, a word is accepted if *all of its runs* are accepting. From the structure of this universal automaton, one can compute a number K with the following property: the environment wins the game as soon as it can enforce at least K visits to the co-Büchi condition. This crucial property makes it possible to turn the game into a K-bounded safety game: the controller's winning condition is now to avoid K visits to the co-Büchi condition. Such a K-bounded safety game is then unfolded to a safety game (the dual of a reachability game), solved in polynomial time.

The bound K and the K-bounded safety game (and *a fortiori* the unfolded safety game) are exponential in the size of the universal co-Büchi automaton (more precisely, $K = n^{2n+3}$ for an automaton with n states), so the worst-case complexity of this approach matches the worst-case complexity of the parity game approach. The crucial difference is that the K-bounded safety game can be solved incrementally: K is an *upper bound* to the number of visits to avoid, so the controller may win while visiting the co-Büchi condition a much smaller number of times k. The incremental algorithm solves the k-bounded safety algorithm, starting from small values of k, and increments it until a winning strategy is found or k eventually exceeds K. While the worst-case complexity remains the same, the incremental approach has proved very efficient in practice, and has been adopted by most tools taking part to the SYNTCOMP competition.

3 Definitions and notations

3.1 Reactive Synthesis

Formally, an instance of the reactive synthesis problem is a triple $(\mathscr{I}, \mathscr{O}, L)$ where $\mathscr{I}$ and $\mathscr{O}$ are two disjoint sets of input and output events respectively, and L is an ω-regular language of infinite words over the alphabet $2^{\mathscr{I} \cup \mathscr{O}}$. For the sake of generality, we assume in this paper that L is given as an ω-automaton.

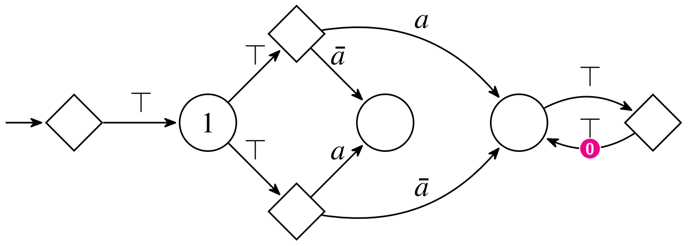


Figure 1: A non-good-for-games arena for Σ^ω.

A *controller* is a function $C : (2^{\mathscr{I}\cup\mathscr{O}})^* \times 2^{\mathscr{I}} \mapsto 2^{\mathscr{O}}$. An infinite word $u = u_0u_1u_2\cdots \in (2^{\mathscr{I}\cup\mathscr{O}})^\omega$ is *consistent* with controller C if, for every $n \in \mathbb{N}$, $u_n \cap \mathscr{O} = C(u_0 \ldots u_{n-1}, u_n \cap \mathscr{I})$. A controller C is said to *enforce* L if every infinite word consistent with C is in L. Given $\mathscr{I}$, $\mathscr{O}$ and L, the reactive synthesis problem asks whether there exists a controller enforcing L, and asks for a witness if the answer is positive.

3.2 ω-Automata

An ω-automaton is a tuple $\mathscr{A} = (Q, \Sigma, \Delta, q_0, F, \phi)$ where:

- Q is a finite set of states;
- Σ is an alphabet (*i.e.* a finite set of letters);
- $q_0 \in Q$ is the initial state;
- $F \subseteq \mathbb{N}$ is a finite set of acceptance marks;
- $\Delta \subseteq Q \times \Sigma \times 2^F \times Q$ is the transition relation;
- ϕ is the acceptance condition (to be detailed below).

A *run* of the automaton is an infinite sequence of consecutive transitions, starting from q_0. The label of a run is the word formed by the concatenation of the letters appearing on its constitutive transitions. Given a run ρ, we define $\mathit{Inf}(\rho) = \{f \in F \mid f \text{ appears on infinitely many transitions of } \rho\}$.

In this paper, we focus on two acceptance conditions: generalized Büchi and parity. In the generalized Büchi setting, a run ρ is accepting if and only if $\mathit{Inf}(\rho) = F$. In other words, a run is accepting if it visits all acceptance marks infinitely often. In the parity setting, a run ρ is accepting if and only if $\max \mathit{Inf}(\rho)$ is odd.

Figure 2a shows a generalized Büchi automaton over the alphabet $2^{\{a,b\}}$. Transitions are labelled by Boolean formulas over the variables $\{a,b\}$. The label $a|b$ of the transition going from state 0 to state 2 is a shorthand that actually represents three different letters: $\{a\}$, $\{b\}$ and $\{a,b\}$. The acceptance marks are denoted by colored numbered bullets on the transitions.

3.3 Games

A *game* is an ω-automaton where the set of states Q is partitioned into two sets Q_A and Q_E. Graphically (see for example Figure 2c), we represent Adam's nodes (Q_A) with diamonds, and Eve's nodes (Q_E) with circles. We refer to the underlying automaton (seen as an automaton and not as a game) as its *arena*. Runs of the arena are called *plays* in the game setting. By convention, the acceptance condition of the arena is the winning condition for Eve in the game: a play is won by Eve if and only if it is accepting in the arena. Otherwise, the play is won by Adam (in particular, finite plays are won by Adam). It is also convenient to partition Σ into Σ_A (letters played by Adam) and Σ_E (letters played by Eve). This partition is done without loss of generality. This restricts Δ so that transitions from Q_A (resp. Q_B) are labelled with letters from Σ_A (resp. Σ_B) only.

A *strategy* for Adam (resp. Eve) is a function mapping finite plays ending in Q_A (resp. Q_E) to a valid transition extending the play. Given a strategy σ for Adam (resp. Eve), a σ-play is a (finite or infinite) play ρ where every transition taken by Adam (resp. Eve), say at position i, is given by $\sigma(\rho_0 \dots \rho_i)$. A *positional* (or *memoryless*) strategy is a strategy whose value depends only on the state in which the given run ends. Given one strategy for each player, say σ_A and σ_E, there is a unique longest play that is both a σ_A-play and a σ_E-play, which is called the *outcome* of σ_A and σ_E. A *winning strategy* for a player is a strategy that makes that player win its outcome against every strategy for the other player. A game is *turn-based* if Δ alternates between Q_A and Q_E (*i.e.* no player plays twice in a row).

4 From Reactive Synthesis to Parity Games

Let $\mathscr{I}$, $\mathscr{O}$ be two sets of input and output signals, and $\mathscr{A}$ be an ω-automaton on the alphabet $2^{\mathscr{I} \cup \mathscr{O}}$. `ltlsynt` solves this reactive synthesis instance by turning $\mathscr{A}$ into a *turn-based deterministic parity game*. We detail the whole process implemented in `ltlsynt`, and show the correctness and completeness of our approach. The method is illustrated with an example presented in Section 4.1.

The first step, called *splitting* (see Section 4.2) separates the input signals from the output signals. This is a mere technicality, but it greatly simplifies the subsequent implementation of parity games. We also believe that it helps understanding the correspondence between winning strategies and controllers.

We then show in the proof of Theorem 1 how a winning strategy in a split arena, with light additional requirements, can be turned into a controller enforcing the corresponding specification.

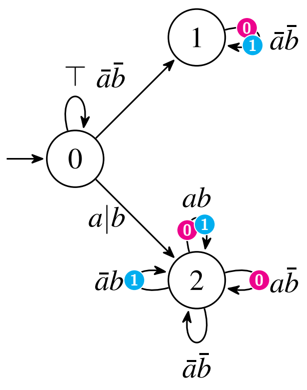
Unfortunately, the converse construction (from a controller to a winning strategy) is not always possible, as illustrated by the following counter-example. Let $\mathscr{I} = \{a\}$, $\mathscr{O} = \{b\}$ and $L = (2^{\mathscr{I} \cup \mathscr{O}})^{\omega}$. Clearly there is a controller enforcing L (in fact any controller enforces L). The game shown on Figure 1 satisfies the hypotheses of Theorem 1, yet it is won by Adam. This comes from the fact that the only way for Eve to resolve the non-determinism in state 1 would be to know in advance Adam's next move. The converse of Theorem 1 only holds if there is some strategy (looking at the past only) for Eve to resolve such non-determinism. This is the precise definition of good-for-games, or history-deterministic, arenas (*cf.* Section 2.2). As we have seen, no practical algorithm is known to produce such arenas, and we resort to determinizing our arena.

In Section 4.3, we prove Theorem 2 which is the converse of Theorem 1. The proof is done for deterministic arenas only, but adapting it to good-for-games arenas would only require minor technical changes.

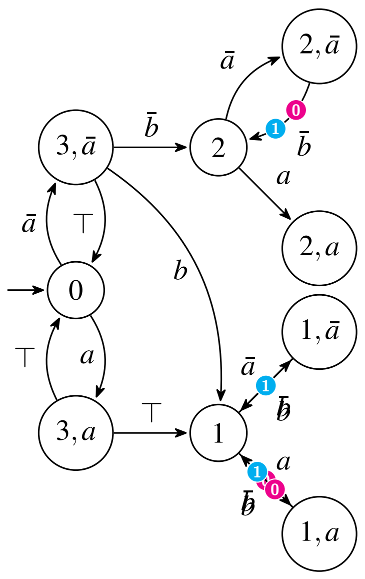
4.1 Running Example

We illustrate our method with an example, which features a single uncontrollable action a and a single controllable action b. The specification states that the controller plays b infinitely often if and only if the environment plays a infinitely often. The corresponding LTL formula is $\mathsf{GF}a \Leftrightarrow \mathsf{GF}b$.

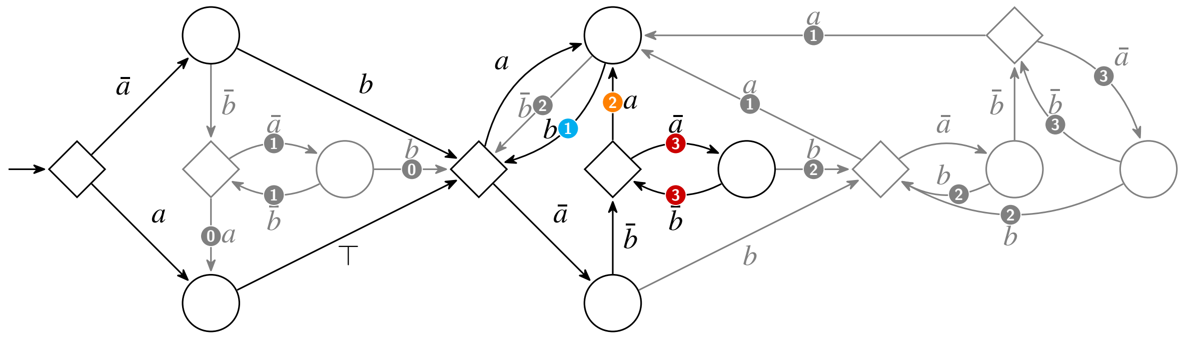
The different steps of the procedure are shown on Figure 2. Figure 2a shows the Transition-based Generalized Büchi Automaton (TGBA) built by Spot from the LTL formula $\mathsf{GF}a \Leftrightarrow \mathsf{GF}b$. The corresponding split automaton is shown on Figure 2b.



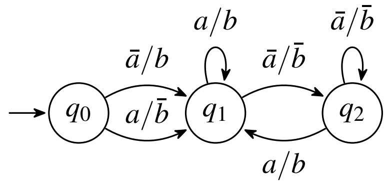
(a) Transition-based Generalized Büchi Automaton for the specification $\mathsf{GF}a \Leftrightarrow \mathsf{GF}b$. A run is accepting if it visits both marks 0 and 1 infinitely often.



(b) The corresponding split automaton. A transition $q \xrightarrow{\ell} q'$ is split into several pairs of the form $q \xrightarrow{a} (q,a) \xrightarrow{\ell \cap 2^{\mathscr{O}}} q'$, one for each $a \in \ell \cap 2^{\mathscr{I}}$.



(c) The corresponding (deterministic) parity game. Eve wins if the minimum priority encountered infinitely often is odd. A winning strategy for Eve is highlighted: transitions not taken by this strategy (and parts of the game that thus become unreachable) are grayed out.



(d) The transducer representing Eve's winning strategy. `ltlsynt` can further reduce it by using standard techniques to reduce the size of finite automata (here q_1 and q_2 would be merged).

Figure 2: The automata produced by `ltlsynt` at different steps of the resolution procedure for $\mathscr{I} = \{a\}$, $\mathscr{O} = \{b\}$ and $L = \mathscr{L}(\mathsf{GF}a \Leftrightarrow \mathsf{GF}b)$.

The determinization of the split automaton as a parity automaton on Figure 2c. The final result of the procedure, a winning strategy for Eve in the form of an automaton, is displayed on Figure 2d. Note that this is the strategy computed by Zielonka's algorithm on the above game. Although it could be further simplified (to an automaton with a single state for instance), `ltlsynt` does not have such simplification capability yet.

4.2 Split Automaton

In order to obtain a turn-based game, we first need to separate input events (played by the environment) from output events (played by the controller). To that end, we define a *split* operation, which applies to words, languages and automata.

Let $u = (u_i)_{i\in\mathbb{N}} \in (2^{\mathscr{I}\cup\mathscr{O}})^\omega$. We define $split_{\mathscr{I},\mathscr{O}}(u) = (v_i)_{i\in\mathbb{N}} \in (2^{\mathscr{I}}2^{\mathscr{O}})^\omega$ as follows: for all $i \in \mathbb{N}$, $v_{2i} = u_i \cap \mathscr{I}$ and $v_{2i+1} = u_i \cap \mathscr{O}$. This operation naturally extends to languages: $split_{\mathscr{I},\mathscr{O}}(L) = \{split_{\mathscr{I},\mathscr{O}}(u) \mid u \in L\}$.

The split of an automaton $\mathscr{A} = (Q, 2^{\mathscr{I}\cup\mathscr{O}}, \Delta, q_0, F)$ is the automaton, noted $split_{\mathscr{I},\mathscr{O}}(\mathscr{A})$, $(Q \cup Q \times 2^{\mathscr{I}}, 2^{\mathscr{I}\cup\mathscr{O}}, \Delta_s, q_0, F)$ where each transition $(q,a,f,q') \in \Delta$ gives, for each $i \in a \cap 2^{\mathscr{I}}$, two transitions in Δ_s: $(q,i,\emptyset,(q,i))$ and $((q,i), a\cap 2^{\mathscr{O}}, f, q')$. It is easy to check that $\mathscr{L}(split_{\mathscr{I},\mathscr{O}}(\mathscr{A})) = split_{\mathscr{I},\mathscr{O}}(\mathscr{L}(\mathscr{A}))$. We also note *split* instead of $split_{\mathscr{I},\mathscr{O}}$ if there is no ambiguity.

We say that an arena is *complete for some player P* if from all states of P, for all letters $a \in \Sigma_P$, there is at least an outgoing transition labelled by a. To complete an arena for P, it suffices to add the missing transitions from every state of Q_P to a new sink state (*i.e.* with no outgoing transitions). Conversely, we say that an arena is *deterministic for a player P* if from all states of P, for all letters $a \in \Sigma_P$, there is at most one outgoing transition labelled by a. Note that the definition of the split automaton ensures that $split_{\mathscr{I},\mathscr{O}}(\mathscr{A})$ is always deterministic for Adam.

Theorem 1. *Let G be a game whose arena A is complete and deterministic for Adam and that recognizes* $split(L)$*. If Eve wins G, then there is a controller enforcing L.*

Proof. Note that the specification to enforce $split(L)$ imposes that G is turn-based. Let σ be a winning strategy (possibly with memory) for Eve in G. From σ we inductively build a controller C enforcing L as follows.

Let $u \in (2^{\mathscr{I}\cup\mathscr{O}})^*$ be a finite history and $i \in 2^{\mathscr{I}}$. If u does not label a σ-play, then we leave C undefined on (u,i). Otherwise, the corresponding σ-play ρ_u is unique: A being deterministic for Adam, his moves are entirely determined by their labels, while Eve's moves are entirely determined by σ. Let q be the state in which ρ_u ends. Again, A being complete and deterministic for Adam, there is a unique transition t labelled i from q. And since σ is a winning strategy, it is necessarily defined on $\rho_u t$. Let o be the label of the transition $\sigma(\rho_u t)$. We define $C(u,i) = o$.

It is clear from this definition that any infinite play u according to C is also the label (up to splitting) of a σ-play in A. This play is therefore won by Eve, meaning that its label $split(\rho)$ is in $split(L)$, which in turn entails that $\rho \in L$. $\square$

4.3 Determinization

Spot implements a determinization procedure from Büchi automata to parity automata, based on an optimized version of Piterman's procedure by Redziejowski [28]. An important feature of this implementation is that both input and output automata have their acceptance marks on transitions rather than on states. This yields fewer states in the determinized automaton. We believe that this is one reason for the efficiency of Spot, compared to other tools that work with state-based acceptance marks.

We now prove the converse of Theorem 1, with the additional requirement that the arena is deterministic for both players (and not only for Adam). As above, we still require that the arena is complete for Adam.

Theorem 2. *Let G be a game whose arena is complete for Adam, deterministic (for both players) and recognizes the language $split(L)$. If there is a controller enforcing L, then Eve wins the game G.*

Proof. Let C be a controller enforcing L. We build a winning strategy (with memory) σ for Eve from C.

Let $(u,i) \in (2^{\mathscr{I}\cup\mathscr{O}})^* \times 2^{\mathscr{I}}$. If u is consistent with C, it can be continued by at least one word in L. A recognizes $split(L)$ and is deterministic, which guarantees that there is a play ρ_u, necessarily unique, labelled by $split(u)$ in A. Let q be the ending state of this play. A being complete for Adam, and deterministic, there is a unique transition t labelled by i from q, going to a state that we call q'. Again, since C enforces L, $u.(i \cup C(u,i))$ can be continued to a word in L. Since A is deterministic and recognizes $split(L)$, there is a (unique) transition t' from q' labelled with $C(u,i)$, and we let $\sigma(\rho_u t) = t'$. Otherwise (*i.e.* if u is not consistent with C), yet ρ_u exists, $\sigma(\rho_u t)$ is left undefined.

It is clear from this definition that any infinite σ-play ρ is labelled by the split of a play consistent with C. Since C enforces L, the label of ρ is in $split(L)$, which means that ρ is an accepting run in A, *i.e.* ρ is won by Eve. □

4.4 Solving Parity Game

Among the various algorithms that have been proposed to solve parity games, we chose to implement the well-known recursive algorithm by Zielonka [32]. This decision was made based on preliminary experiments with the framework PGSolver [14], that convinced us that Zielonka's algorithm appears to be the most efficient in practice. A recent, independent, experimental study [6] confirms that attractor-based algorithms, and in particular Zielonka's algorithm, appear to be the most efficient in practice.

As shown in Section 6, the resolution of parity games does not seem to be the bottleneck of our approach. We therefore did not invest more efforts into implementing other parity game solving algorithms so far. Should we improve the performance of our parity game algorithms, using an optimized, specialized library such as Oink [6] would probably be easier and more efficient than re-implementing state-of-the-art algorithms ourselves.

For the sake of completeness, we mention that `ltlsynt`features an experimental version of the quasi-polynomial algorithm by Calude *et al.* [3], but this implementation has not been extensively tested, optimized, nor evaluated.

5 Using `ltlsynt`

`ltlsynt` comes as a binary in the Spot library, starting from version 2.5. `ltlsynt` expects three input elements: a set of input propositions $\mathscr{I}$, a set of output propositions $\mathscr{O}$ and a specification, as an LTL or PSL formula [1]. The possible formats are those supported by Spot: we refer the reader to Spot documentation for details.

In SYNTCOMP, the specification is described as a Moore or Mealy machine, using the TLSF format [19]. Such specifications are translated to LTL formulas, thanks to the tool `syfco` [1] before they are

[1]`https://github.com/reactive-systems/syfco/releases`

given to `ltlsynt`.

```
ltlsynt --ins=a --outs=b -f 'GFa <-> GFb' [--realizability]
```

`ltlsynt` outputs “REALIZABLE”, followed by a winning strategy in HOA format [2], or “UNREALIZABLE”. An option allows to check for realizability only, disabling the computation of the winning strategy.

6 Experimental assessment

We experimentally evaluate `ltlsynt` in two steps. We first recall the results obtained at SYNTCOMP’2017, as it provides an independent evaluation of our tool compared to other state-of-the-art tools. We only highlight the main facts about the performance of `ltlsynt`. We also report on more detailed experiments we undertook to better understand the reasons of the performance of `ltlsynt`, and identify its weaknesses.

6.1 Report from SYNTCOMP’2017

`ltlsynt` has participated to SYNTCOMP’2017, and ranked second in the TLSF sequential realizability track. We reproduce here some results from the competition highlighting the overall performance of `ltlsynt` compared to the other competing tools. Detailed results can be found in the report on the competition [17].

The competition aims two problems: the *realizability* problem, asking whether the controller is realizable, and the *synthesis* problem, that further requires to compute a satisfying controller if one exists.

Table 1 summarizes the results of the sequential TLSF track of the competition (the track in which `ltlsynt` was engaged). The realizability table shows for each participating tool the number of instances that it is able to solve within the 3600s time limit, as well as the number of instances that no other tool solve (“Unique” column). The synthesis table additionally reports the number of solutions that could be model-checked (hence certified as correct), as well as a quality score, based on the size of the computed controller with respect to reference solutions. The smaller the circuits, the higher the quality. The benchmark consists of 244 instances. Some tools participate multiple times with different configurations: for those tools, the name of the configuration is indicated within parentheses.

Note that PARTY-Elli and `ltlsynt` participated for the first time: they both solve more instances than the best tools from previous editions. This is already a great confirmation of the efficiency of `ltlsynt`. A more attentive look at the results for synthesis reveals that `ltlsynt` ranks second in number of instances solved, but its quality score remains quite low: this indicates that the synthesized circuits tend to be larger than the reference solutions. Despite its good performance at solving the synthesis problem, the circuit synthesis itself deserves to be optimized. The version submitted in 2017 makes no attempt at reducing the size of the winning strategy. We hope that the recent implementation of this capability will result in higher quality scores in the next editions of the competition.

6.2 Evaluation

Spot is widely recognized as one of the best existing tools to translate LTL formulae to ω-automata [29]. Besides this translation, Spot is indeed able to perform syntactical rewritings and simplifications of LTL formulas (as a pre-processing step), and also to simplify ω-automata, mainly through signature-based simulation algorithms (as a post-processing step) [7]. Our first impression is that these capabilities allow

Table 1: Summary of the results of the sequential TLSF track of SYNTCOMP'2017, for realizability and synthesis.

Realizability			Synthesis				
Tool	Solved	Unique	Tool	Solved	Checked	Unique	Quality
Party-Elli (kid)	218	7	Party-Elli (kid)	220	200	4	219
`ltlsynt`	195	3	`ltlsynt`	195	182	3	180
BoSy (spot)	181	0	BoSy (spot)	181	181	3	298
BoSy (ltl3ba)	172	0	Party-Elli (int)	167	167	0	249
Party-Elli (int)	169	0	BoSy (ltl3ba)	165	165	0	277
BoWser	165	0	Party-Elli (bool)	163	163	1	222
Party-Elli (bool)	164	0	BoWser (c0)	162	162	0	273
Acacia4Aiger	142	4	BoWser (c1)	155	155	0	260
			Acacia4Aiger	127	17	2	91
			BoWser (c2)	93	93	0	141

Spot to produce smaller automata, and subsequentially smaller games, than the other competing tools at SYNTCOMP'2017. But as we can see on Table 1, the version of Bosy that relies on Spot for the translation of LTL specifications to automata performs worse than `ltlsynt`.

We thus decided to undertake more detailed experiments, to better understand the strengths and weaknesses of `ltlsynt`. First, we have measured the relative time taken by each of the four steps of the synthesis resolution (namely: translation of the formula, splitting of the resulting automaton, determinization of the split automaton, and resolution of the obtained parity game). Our primary hope is to identify a critical, bottleneck, step among the four, on which we should concentrate future efforts. Second, we measured the same steps, but with many optimizations deactivated: no syntactic simplifications, no simulation-based reductions, use of state-based acceptance condition ...The goal is to understand whether the extra computation effort put in the first steps to obtain small arenas is worth the time saved in the resolution of the parity game.

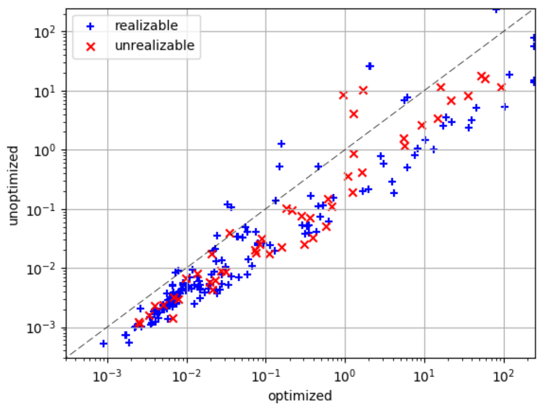
Experiments use an instrumented version of the `ltlsynt` binary, packaged in the development version of Spot 2.5.4, which differs from the binary submitted at the competition. Especially, the tested version features optimizations to the determinization and Zielonka's algorithms that have been implemented after the submission to SYNTCOMP'2017. Experiments were run on a machine with an Intel Core i5-6260U CPU (4 cores at 1.80GHz) and 8GB of RAM, running a Debian 9. The tool was compiled using `g++-6.3.0`. We used no memory confinement, and a time confinement of 120 seconds. The running times were measured using high-resolution timers from the C++11 standard library. The benchmarks that we use are those used in the SYNTCOMP'2017 competition, which features 244 instances.

Raw data, as well as scripts to run the experiments can be found at `https://lrde.epita.fr/~max/synt2018-ltlsynt.zip`. The archive features a Dockerfile to build a container to run the experiments. It also contains the raw data of our own experiments.

We first compare the overall performance of the two versions (with and without the optimizations) of `ltlsynt`. Results are summarized in Figure 3. More precisely, Figure 3a presents the number of instances and the par-2 score of each versions, while Figure 3b plots compares the performance of the two versions instance by instance. Each point in this scatterplot represents one instance, whose coordinates correspond to the running times of the two versions; points below the diagonal (the dashed line) are those for which the unoptimized version runs faster. Note that the scale is logarithmic; the dotted line represents the place where the unoptimized version runs 5 times faster than the optimized one.

	unoptimized	optimized
# instances solved	200	197
par-2 time (s)	44.9	49.9

(a) Comparison of the number of instances solved and of the par-2 score.



(b) Comparison of the runtimes of the optimized and unoptimized versions of `ltlsynt`.

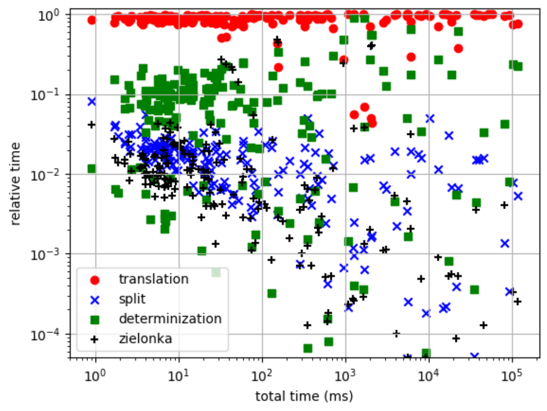
Figure 3: Comparison of the two versions of `ltlsynt`.

We observe that the unoptimized version solves 3 more instances within the time limit than the optimized version. More significantly, the PAR-2 score of the unoptimized version is 10% lower than the PAR-2 score of the optimized version. The scatterplot gives a stronger trend: a large majority of instances are solved faster, by a factor between 1 and 10, with the unoptimized version. There does not seem to be any significant difference between realizable and unrealizable instances.

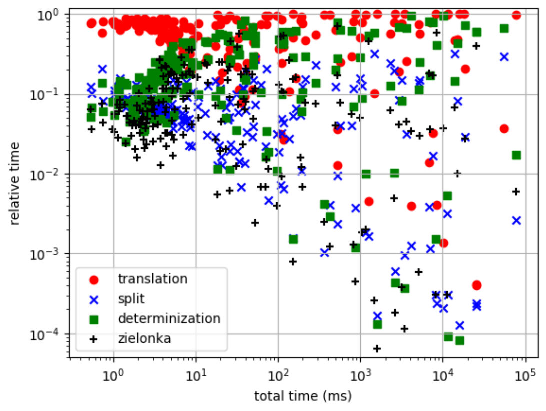
These results indicate that spending time to reduce the size of the arena does not pay off. This conclusion is somehow surprising: we expected that smaller automata would yield smaller deterministic automata, hence smaller arenas. Among the simplifications performed by Spot, one tests for implications between subformulae of the formula to translate. These implication tests end up, in the worst-case, translating the formulae to automata and testing language inclusion. This kind of test is computationnally expensive, and usually yields ludicrous gains on automaton size.

The complete translation process of Spot features several steps, each with several level of optimizations to be fine-tuned. We have only the three pre-defined settings for the translation process, but there is a large number of variants that remain to be tested.

We now turn to the time spent in each step of the process. Results are reported graphically in Figure 4. We observe that the translation is the step taking the largest portion of the time, both in the optimized and unoptimized versions. More precisely, splitting and Zielonka's algorithm took in total less than 10% of the total running time of the optimized version, while the determinization takes between 0.1% and 20% for the majority of instances. Clearly the bottleneck of the optimized version is the translation step, but this is not very suprising given the amount of simplifications that take place during it. The picture is a bit more contrasted for the unoptimized version: splitting may take up to 20% of the time, but Zielonka's algorithm is most of the time below 10%. Translation and determinization seem to share almost equally the rest of the time, except for instances solved instantly (in less than 10ms), where translation dominates.



(a) Optimized version



(b) Unoptimized version

Figure 4: Relative runtime of each step of the resolution.

7 Conclusion and Future Work

We have presented our approach and our tool for reactive synthesis from LTL specifications. Contrary to most other tools, we solve the synthesis problem by translation to a parity game that we solve with a recursive algorithm. Due to the double-exponential upper bound for the size of the parity games, this approach has long been thought to be impractical. Our tool, that ranked second in the SYNTCOMP'2017 competition, demonstrates the practicality of this approach.

The journey towards practical reactive synthesis goes on. The other approach to reactive synthesis, by reduction to safety games, supports an incremental resolution. We think that such an incremental approach could be designed in the parity game approach. One idea would be to remove non-determinism of the arena incrementally, so as to limit the blow-up induced by the determinization.

There also seems that translation of LTL formulae to ω-automata could be improved. Recent work [12] proposes a translation algorithm that produces deterministic automata by design. The authors demonstrate that this approach may outperform Spot, so we intend to implement their approach in Spot.

References

[1] (2004): *Property Specification Language Reference Manual v1.1*. Accellera. `http://www.eda.org/vfv/`.

[2] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker & Jan Strejček (2015): *The Hanoi Omega-Automata Format*. In: *Proceedings of the 27th International Conference on Computer Aided Verification (CAV'15)*, *Lecture Notes in Computer Science* 9206, Springer, pp. 479–486, doi:10.1007/978-3-319-21690-4_31.

[3] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li & Frank Stephan (2017): *Deciding Parity Games in Quasipolynomial Time*. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, ACM, New York, NY, USA, pp. 252–263, doi:10.1145/3055399.3055409.

[4] Krishnendu Chatterjee, Monika Henzinger & Veronika Loitzenbauer (2015): *Improved Algorithms for One-Pair and k-Pair Streett Objectives*. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, IEEE Computer Society, pp. 269–280, doi:10.1109/LICS.2015.34.

[5] Thomas Colcombet (2012): *Forms of determinism for automata*. In: *STACS'12 (29th Symposium on Theoretical Aspects of Computer Science)*, 14, LIPIcs, pp. 1–23.

[6] Tom van Dijk (2018): *Oink: An Implementation and Evaluation of Modern Parity Game Solvers*. In Dirk Beyer & Marieke Huisman, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, *Lecture Notes in Computer Science* 10805, Springer, pp. 291–308, doi:10.1007/978-3-319-89960-2_16.

[7] Alexandre Duret-Lutz (2014): *LTL translation improvements in Spot 1.0*. *IJCCBS* 5(1/2), pp. 31–54, doi:10.1504/IJCCBS.2014.059594.

[8] 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: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, Proceedings*, *Lecture Notes in Computer Science* 9938, pp. 122–129, doi:10.1007/978-3-319-46520-3_8.

[9] Stefan Dziembowski, Marcin Jurdziński & Igor Walukiewicz (1997): *How Much Memory is Needed to Win Infinite Games?* In: *12th Annual IEEE Symposium on Logic in Computer Science, LICS*, IEEE Computer Society, pp. 99–110, doi:10.1109/LICS.1997.614939.

[10] E. A. Emerson & C. S. Jutla (1991): *Tree Automata, Mu-Calculus and Determinacy*. In: *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, IEEE Computer Society, Washington, DC, USA, pp. 368–377, doi:10.1109/SFCS.1991.185392.

[11] E. Allen Emerson & Charanjit S. Jutla (1988): *The Complexity of Tree Automata and Logics of Programs (Extended Abstract)*. In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, IEEE Computer Society, pp. 328–337, doi:10.1109/SFCS.1988.21949.

[12] Javier Esparza, Jan Křetínský, Jean-François Raskin & Salomon Sickert (2017): *From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Uppsala, Sweden, Proceedings, Part I*, *Lecture Notes in Computer Science* 10205, pp. 426–442, doi:10.1007/978-3-662-54577-5_25.

[13] John Fearnley, Sanjay Jain, Sven Schewe, Frank Stephan & Dominik Wojtczak (2017): *An Ordered Approach to Solving Parity Games in Quasi Polynomial Time and Quasi Linear Space*. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, ACM, New York, NY, USA, pp. 112–121, doi:10.1145/3092282.3092286.

[14] Oliver Friedmann & Martin Lange: *PGSolver*. Available at `https://github.com/tcsprojects/pgsolver`.

[15] Hugo Gimbert & Rasmus Ibsen-Jensen (2017): *A short proof of correctness of the quasi-polynomial time algorithm for parity games*. *CoRR* abs/1702.01953. Available at `http://arxiv.org/abs/1702.01953`.

[16] Thomas A Henzinger & Nir Piterman (2006): *Solving games without determinization*. In: *International Workshop on Computer Science Logic*, Springer, pp. 395–410.

[17] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur & Leander Tentrup (2017): *The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants & Results*. In Dana Fisman & Swen Jacobs, editors: Proceedings Sixth Workshop on *Synthesis*, Heidelberg, Germany, 22nd July 2017, *Electronic Proceedings in Theoretical Computer Science* 260, Open Publishing Association, pp. 116–143, doi:10.4204/EPTCS.260.10.

[18] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The first reactive synthesis competition (SYNTCOMP 2014)*. *International Journal on Software Tools for Technology Transfer* 19(3), pp. 367–390, doi:10.1007/s10009-016-0416-3.

[19] Swen Jacobs, Felix Klein & Sebastian Schirmer (2016): *A High-Level LTL Synthesis Format: TLSF v1.1*. In Ruzica Piskac & Rayna Dimitrova, editors: *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, *EPTCS* 229, pp. 112–132, doi:10.4204/EPTCS.229.10.

[20] Marcin Jurdziński (2000): *Small Progress Measures for Solving Parity Games*. In Horst Reichel & Sophie Tison, editors: *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, *Lecture Notes in Computer Science* 1770, Springer, pp. 290–301, doi:10.1007/3-540-46541-3_24.

[21] Marcin Jurdziński & Ranko Lazić (2017): *Succinct progress measures for solving parity games*. In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1–9, doi:10.1109/LICS.2017.8005092.

[22] Marcin Jurdziński, Mike Paterson & Uri Zwick (2008): *A Deterministic Subexponential Algorithm for Solving Parity Games*. *SIAM J. Comput.* 38(4), pp. 1519–1532, doi:10.1137/070686652.

[23] Denis Kuperberg & Michał Skrzypczak (2015): *On determinisation of good-for-games automata*. In: *International Colloquium on Automata, Languages, and Programming*, Springer, pp. 299–310.

[24] Orna Kupferman (2006): *Avoiding Determinization*. In: *21st Annual IEEE Symposium on Logic in Computer Science, LICS*, IEEE Computer Society, Washington, DC, USA, pp. 243–254, doi:10.1109/LICS.2006.15.

[25] Donald A. Martin (1975): *Borel Determinacy*. *Annals of Mathematics* 102(2), pp. 363–371.

[26] Matthias Mnich, Heiko Röglin & Clemens Rösner (2016): *New Deterministic Algorithms for Solving Parity Games*. In Evangelos Kranakis, Gonzalo Navarro & Edgar Chávez, editors: *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, *Lecture Notes in Computer Science* 9644, Springer, pp. 634–645, doi:10.1007/978-3-662-49529-2_47.

[27] Amir Pnueli & Roni Rosner (1989): *On the synthesis of a reactive module*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 179–190.

[28] Roman R. Redziejowski (2012): *An Improved Construction of Deterministic Omega-automaton Using Derivatives*. *Fundamenta Informaticae* 119(3-4), pp. 393–406.

[29] Kristin Y Rozier & Moshe Y Vardi (2010): *LTL Satisfiability Checking*. *International journal on software tools for technology transfer* 12(2), pp. 123–137.

[30] Sven Schewe (2007): *Solving Parity Games in Big Steps*. In Vikraman Arvind & Sanjiva Prasad, editors: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*, *Lecture Notes in Computer Science* 4855, Springer, pp. 449–460, doi:10.1007/978-3-540-77050-3_37.

[31] Sven Schewe (2009): *Tighter Bounds for the Determinisation of Büchi Automata*. In Luca de Alfaro, editor: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, York, UK. Proceedings*, *Lecture Notes in Computer Science* 5504, Springer, pp. 167–181, doi:10.1007/978-3-642-00596-1_13.

[32] Wiesław Zielonka (1998): *Infinite games on finitely coloured graphs with applications to automata on infinite trees*. *Theoretical Computer Science* 200(1), pp. 135 – 183, doi:http://dx.doi.org/10.1016/S0304-3975(98)00009-7.