Dissecting `ltlsynt`

Florian Renkin[1*], Philipp Schlehuber-Caissier[1], Alexandre Duret-Lutz[1] and Adrien Pommellet[1]

[1]EPITA Research Laboratory, EPITA, 14–16 rue Voltaire, 94270 Kremlin-Bicêtre, France.

*Corresponding author(s). E-mail(s): renkin@lrde.epita.fr;
Contributing authors: philipp@lrde.epita.fr; adl@lrde.epita.fr; adrien@lrde.epita.fr;

Abstract

`ltlsynt` is a tool for synthesizing a reactive circuit satisfying a specification expressed as an LTL formula. `ltlsynt` generally follows a textbook approach: the LTL specification is translated into a parity game whose winning strategy can be seen as a Mealy machine modeling a valid controller. This article details each step of this approach, and presents various refinements integrated over the years. Some of these refinements are unique to `ltlsynt`: for instance, `ltlsynt` supports multiple ways to encode a Mealy machine as an AIG circuit, features multiple simplification algorithms for the intermediate Mealy machine, and bypasses the usual game-theoretic approach for some subclasses of LTL formulas in favor of more direct constructions.

Data Availability Statement: Instructions to reproduce the benchmarks presented in this article are available from https://www.lrde.epita.fr/~frenkin/fmsd22/artifact

Keywords: reactive synthesis, Mealy machines, parity automata, parity games, LTL formulas

1 Introduction

Program synthesis is a well-established formal method: given a logical specification of a system, it allows one to automatically generate a provably correct

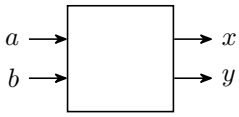


Fig. 1: A reactive controller, seen as a black box that reads some input signals (here a, b) to produce some output signals (here x, y).

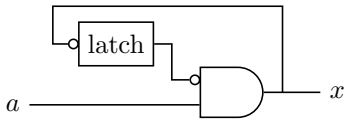


Fig. 2: An And-Inverter Graph where the latch is used to remember if a has been false. Output x is true as long as a remains true, and becomes continuously false on the first occurrence of $\bar{a}$.

implementation. It can be applied to *reactive controllers* (Fig. 1), that is, circuits that read an input stream of Boolean valuations (here, over Boolean variables a and b) and simultaneously produce a matching output stream of valuations (here, over x and y). The `ltlsynt` tool that we shall discuss solves this problem when the specification is given as a Linear-time Temporal Logic formula constraining the input and output signals over time. If this specification is realizable, that is, if it is possible to build a controller for it, then the output is expected as an *And-Inverter Graph* (AIG), that is, a circuit built from *and gates* ⊐, *negations* ○, and *latches* □ that delay their input by one tick, and output 0 initially.

For instance, assuming a single input signal a and a single output signal x, the specification formula $a \leftrightarrow \mathsf{F}(x)$, which states that a should hold initially if and only if x holds eventually, could be satisfied by constructing a controller such that the Boolean output value x is continuously equal to the first value of the Boolean input a. However, many other controllers could satisfy this formula. For instance, in Fig. 2, signal x holds until the first $\bar{a}$ is received, and remains false after that occurrence. Among possible choices, we are interested in producing a small controller in terms of AIG size (number of gates and latches).

To obtain the AIG controller, we use an automata-theoretic approach whose main steps are the following:

- Convert the specification LTL formula into a deterministic parity automaton (Fig. 3).
- Knowing that output signals are controllable while input signals are not, convert this parity automaton into a parity game, where player 0 (the environment) chooses the input signals, and player 1 (the controller) chooses the output signals. In addition to being deterministic, this game has to be input complete, i.e., the environment must always be able to select any valuation of the input signals (Fig. 4). The specification is realizable if player 1 has a strategy to respond to any input stream while satisfying the parity acceptance condition.
- A winning strategy for player 1 can be seen as a Mealy machine (Fig. 5). We use incompletely specified generalized Mealy machines to capture some of the freedom provided by the strategy as far as output selection is concerned.

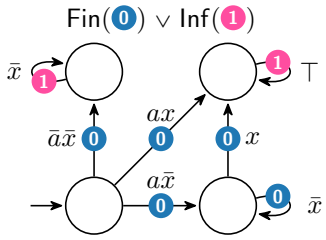


Fig. 3: A deterministic *max odd* parity automaton for $a \leftrightarrow \mathsf{F}(x)$. An infinite run is accepting if the maximal color it visits infinitely often is odd.

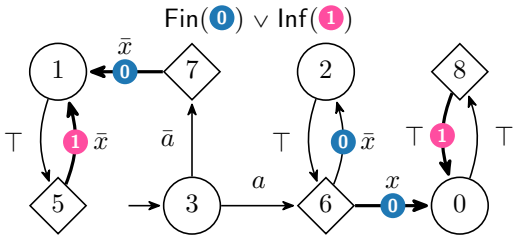


Fig. 4: The two-player parity game interpretation of the automaton of Fig. 3. Player 0 plays from round nodes and selects input signals; player 1 plays from square nodes and selects output signals. The game is winning for player 1 if, regardless of the choices made by player 0, player 1 can force the infinite play to satisfy the *parity max odd* acceptance condition. This is the case here if player 1 always selects the thick transitions.

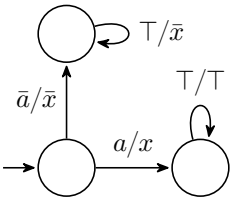


Fig. 5: The winning strategy of Fig. 4, seen as an incompletely specified Mealy machine.

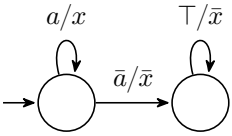


Fig. 6: A reduced Mealy machine, for the winning strategy of Fig. 4. Note how the states of this machine are used to remember if $\bar{a}$ has been seen at some point. This machine can be encoded into the AIG of Fig. 2.

- The above Mealy machine is then reduced (Fig. 6), taking advantage of that freedom if possible.
- Finally the Mealy machine is encoded in the AIGER format, to represent the controller as an AIG (Fig. 2).

The above steps are a mere outline of the procedure implemented in our tool, called `ltlsynt`, and distributed along with the Spot library (Duret-Lutz et al, 2022), where most of the operations are implemented. Figure 7 gives a more detailed picture by including two optimizations: a decomposition of the specification into multiple sub-specifications (we refer the reader to Finkbeiner et al (2021) for more details about this technique), and a shortcut to the "standard" construction for some LTL formulas.

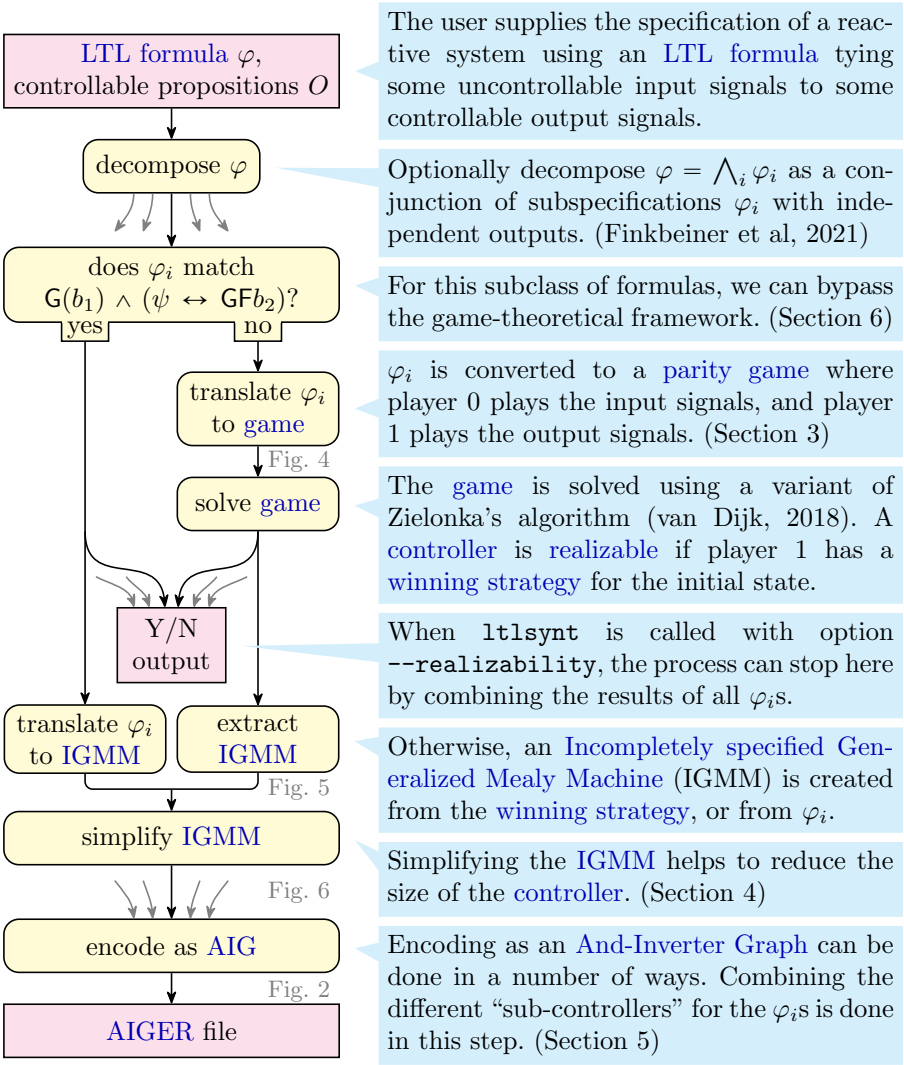


Fig. 7: General outline of the process used by `ltlsynt` to solve a reactive synthesis problem.

The rest of this article is organized as follows. We define LTL, automata, parity games, and Mealy machines in Section 2. The rest of the sections delve into specific steps of this pipeline: Section 3 explains how we convert LTL formulas into deterministic parity automata, Section 4 discusses different options for reducing incompletely specified Mealy machines, Section 5 discusses several ways to encode our Mealy machines into an AIG. Section 6 shows how to bypass the "standard" construction for some subclass of LTL formulas. Finally we evaluate some of these choices in Section 7.

2 Concepts

2.1 Valuations and Cubes

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers, and let $\mathbb{B} = \{\top, \bot\}$ denote the set of Boolean values. Given a set X of propositions (i.e., Boolean variables), a *valuation* is a function from X to $\mathbb{B}$. Let $\mathbb{B}^X$ be the set of all possible valuations on X, and let $2^{\mathbb{B}^X}$ be its set of subsets. Any element of $2^{\mathbb{B}^X}$ can be expressed as a Boolean formula over X, so we shall often represent valuations or set of valuations as Boolean formulas over X. The negation of proposition p is denoted $\bar{p}$. We overload $\top$ (resp. $\bot$) to denote the Boolean formula that is always true (resp. false), or equivalently the set $\mathbb{B}^X$ (resp. $\varnothing$), and assume that X is clear from the context.

A *cube* is a (possibly empty) conjunction of propositions or their negations (i.e., literals). As an example, given three propositions $X = \{a, b, c\}$, the cube $a \wedge \bar{b}$, usually written $a\bar{b}$, stands for the set of valuations in which a is true and b is false, i.e., $\{a\bar{b}c, a\bar{b}\bar{c}\}$. Let $\mathbb{K}^X$ stand for the set of all cubes over X. $\mathbb{K}^X$ contains the cube $\top$ (the empty conjunction), that stands for the set of all possible valuations over X. Note that any set of valuations can be represented as a disjunction of disjoint cubes.

2.2 Linear-Time Temporal Logic

Given an alphabet Σ, we use Σ^ω to denote the set of infinite words over Σ. For a word $\pi = \pi_0\pi_1\pi_2 \ldots \in \Sigma^\omega$ we note π_i its letter at position $i \in \mathbb{N}$. We use $\pi_{i..}$ to denote the suffix of π starting at position i, in other words $\pi_i\pi_{i+1}\pi_{i+2} \ldots = \pi_{i..}$.

A set of words $L \subseteq \Sigma^\omega$ is called a language. We note $L^\complement = \Sigma^\omega \setminus L$ its complement.

In the sequel, we work with an alphabet that contains valuations over a set of input and output propositions (representing the input and output signals of our controller). Let I and O be two disjoint sets of input and output propositions. A *trace* is an infinite word $\pi \in (\mathbb{B}^{I \cup O})^\omega$ over valuations of $I \cup O$. We use symbol $\cup$ to indicate that this is a disjoint union.

A classical formalism to express specifications on traces is LTL:

Definition 1 (LTL) A *Linear-time Temporal Logic* formula is built from the following grammar:

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \odot \varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \mathsf{X}\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi$$

where $p \in I \cup O$ is any proposition, and $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ represents any binary Boolean operator ($\oplus$ is exclusive or).

Parentheses are used for grouping, but we omit them from the above grammar and the upcoming ones for simplicity.

Definition 2 (Semantics of LTL) For a trace $\pi \in (\mathbb{B}^{I \cup O})^\omega$ and an LTL formula φ, we say that π satisfies φ, denoted $\pi \vDash \varphi$, and define this relation inductively as follows:

$$
\begin{aligned}
\pi &\vDash \top \Leftrightarrow \top \\
\pi &\vDash \bot \Leftrightarrow \bot \\
\pi &\vDash \rho \Leftrightarrow \pi_0(\rho) = \top \\
\pi &\vDash \neg\varphi \Leftrightarrow \neg(\pi \vDash \varphi) \\
\pi &\vDash \varphi \odot \psi \Leftrightarrow (\pi \vDash \varphi) \odot (\pi \vDash \psi) \\
\pi &\vDash \mathsf{X}\varphi \Leftrightarrow \pi_{1..} \vDash \varphi \\
\pi &\vDash \mathsf{F}\varphi \Leftrightarrow \exists i \in \mathbb{N},\ \pi_{i..} \vDash \varphi \\
\pi &\vDash \mathsf{G}\varphi \Leftrightarrow \forall i \in \mathbb{N},\ \pi_{i..} \vDash \varphi \\
\pi &\vDash \varphi\,\mathsf{U}\,\psi \Leftrightarrow \exists i \in \mathbb{N},\ \pi_{i..} \vDash \psi \text{ and } \forall j \in \{0,\dots,i-1\},\pi_{j..} \vDash \varphi \\
\pi &\vDash \varphi\,\mathsf{R}\,\psi \Leftrightarrow \begin{cases} \forall i \in \mathbb{N},\ \pi_{i..} \vDash \psi \text{ or} \\ \exists i \in \mathbb{N},\ \pi_{i..} \vDash \varphi \text{ and } \forall j \leqslant i,\ \pi_{j..} \vDash \psi \end{cases}
\end{aligned}
$$

In the above, $\rho \in I \cup O$ is a proposition, φ and ψ are LTL formulas, and $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ is any binary Boolean operator.

The language of a formula φ is the set of traces that satisfy it: $\mathcal{L}(\varphi) = \{\pi \in (\mathbb{B}^{I \cup O})^\omega \mid \pi \vDash \varphi\}$. Two formulas φ and ψ are said to be *equivalent* if they have the same language, and we write $\varphi \equiv \psi$.

We now define some useful subclasses of LTL formulas.

Definition 3 (Some subclasses of LTL) Consider the following grammar rules, where $p \in I \cup O$ is any proposition, $\ominus \in \{\wedge, \vee, \leftrightarrow, \oplus\}$ is any commutative Boolean operator, $\odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ is any binary Boolean operator, and φ is any LTL formula.

$$
\begin{aligned}
\varphi_N &::= \top \mid \bot \mid p \mid \neg p \mid \varphi_N \ominus \varphi_N \mid \mathsf{F}\varphi_N \mid \mathsf{G}\varphi_N \mid \mathsf{X}\varphi_N \mid \varphi_N\,\mathsf{U}\,\varphi_N \mid \varphi_N\,\mathsf{R}\,\varphi_N \\
\varphi_X &::= \top \mid \bot \mid p \mid \neg\varphi_X \mid \varphi_X \odot \varphi_X \mid \mathsf{X}\varphi_X \\
\varphi_G &::= \varphi_X \mid \neg\varphi_S \mid \varphi_G \wedge \varphi_G \mid \varphi_G \vee \varphi_G \mid \varphi_S \rightarrow \varphi_G \mid \mathsf{X}\varphi_G \mid \mathsf{F}\varphi_G \mid \varphi_G\,\mathsf{U}\,\varphi_G \\
\varphi_S &::= \varphi_X \mid \neg\varphi_G \mid \varphi_S \wedge \varphi_S \mid \varphi_S \vee \varphi_S \mid \varphi_G \rightarrow \varphi_S \mid \mathsf{X}\varphi_S \mid \mathsf{G}\varphi_S \mid \varphi_S\,\mathsf{R}\,\varphi_S \\
\varphi_O &::= \varphi_G \mid \varphi_S \mid \neg\varphi_O \mid \varphi_O \odot \varphi_O \mid X\varphi_O \mid \varphi_O\,\mathsf{U}\,\varphi_G \mid \varphi_O\,\mathsf{R}\,\varphi_S \\
\varphi_P &::= \varphi_O \mid \neg\varphi_R \mid \varphi_P \odot \varphi_P \mid \mathsf{X}\varphi_P \mid \mathsf{F}\varphi_P \mid \varphi_P\,\mathsf{U}\,\varphi_P \mid \varphi_P\,\mathsf{R}\,\varphi_S \\
\varphi_R &::= \varphi_O \mid \neg\varphi_P \mid \varphi_R \odot \varphi_R \mid \mathsf{X}\varphi_R \mid \mathsf{G}\varphi_R \mid \varphi_R\,\mathsf{U}\,\varphi_G \mid \varphi_R\,\mathsf{R}\,\varphi_R \\
\varphi_\mu &::= \bot \mid \top \mid \neg\varphi_\nu \mid \varphi_\mu \wedge \varphi_\mu \mid \varphi_\mu \vee \varphi_\mu \mid \mathsf{X}\varphi_\mu \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi_\mu \mid \varphi\,\mathsf{U}\,\varphi_\mu \mid \top\,\mathsf{U}\,\varphi \mid \varphi_\mu\,\mathsf{R}\,\varphi_\mu \\
\varphi_\nu &::= \bot \mid \top \mid \neg\varphi_\mu \mid \varphi_\nu \wedge \varphi_\nu \mid \varphi_\nu \vee \varphi_\nu \mid \mathsf{X}\varphi_\nu \mid \mathsf{F}\varphi_\nu \mid \mathsf{G}\varphi \mid \varphi_\nu\,\mathsf{U}\,\varphi_\nu \mid \varphi\,\mathsf{R}\,\varphi_\nu \mid \bot\,\mathsf{R}\,\varphi \\
\varphi_\xi &::= \bot \mid \top \mid \neg\varphi_\xi \mid \varphi_\xi \odot \varphi_\xi \mid \mathsf{F}\varphi_\nu \mid \mathsf{G}\varphi_\mu \mid \mathsf{F}\varphi_\xi \mid \mathsf{G}\varphi_\xi \mid \mathsf{X}\varphi_\xi \mid \varphi\,\mathsf{U}\,\varphi_\xi \mid \varphi\,\mathsf{R}\,\varphi_\xi
\end{aligned}
$$

Formulas produced by rule φ_N are said to be in *negative normal form* (NNF): i.e., negation is only applied to atomic propositions, and implication is not used.
Formulas produced by rule φ_X are *LTL(X) formulas*, i.e., LTL formulas where the only temporal operator used is X.

Formulas produced by rules φ_G, φ_S, φ_O, φ_P, φ_R are called respectively *syntactic guarantee formulas*, *syntactic safety formulas*, *syntactic obligation formulas*, *syntactic persistence formulas* and *syntactic recurrence formulas* (Černá and Pelánek, 2003). Formulas produced by rules φ_μ, φ_ν, and φ_ξ are called respectively *pure eventuality formulas*, *pure universality formulas*, and *suspendable formulas* (Babiak et al, 2013).

Theorem 2.1 (Folklore) *Any LTL formula can be transformed into an equivalent NNF formula by pushing negations inward, applying rules such as* $\neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \equiv (\neg\varphi_1)\,\mathsf{R}\,(\neg\varphi_2)$, $\neg(\varphi_1 \leftrightarrow \varphi_2) \equiv (\neg\varphi_1) \oplus (\neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv (\neg\varphi_1) \vee \varphi_2$, *etc.*

Theorem 2.2 (Etessami and Holzmann (2000); Babiak et al (2013)) *Any pure eventuality* φ *satisfies* $\varphi \equiv \mathsf{F}\varphi$. *Any pure universality* φ *satisfies* $\varphi \equiv \mathsf{G}\varphi$. *Any suspendable formula* φ *satisfies* $\varphi \equiv \mathsf{G}\varphi \equiv \mathsf{F}\varphi \equiv \mathsf{X}\varphi$.

The sets of syntactic guarantee, syntactic safety, syntactic obligation formulas, syntactic recurrence formulas and syntactic persistence formulas are related to the temporal hierarchy of Manna and Pnueli (1990) in a way that has been discussed by Černá and Pelánek (2003). However, we first need to define automata before we can state those important results.

2.3 Emerson-Lei Automata

It is a well-known result that Muller automata can recognize the set of all traces satisfying a given LTL formula and thus, may prove extremely useful in the synthesis process. *Emerson-Lei Automata* were defined (Emerson and Lei, 1987) and named (Safra and Vardi, 1989) in the 80s; they provide a way to describe a Muller acceptance condition using a positive Boolean formula over sets of states that must be visited finitely or infinitely often. Below we define the transition-based version of those automata, as used in the *Hanoi Omega-Automata Format* (Babiak et al, 2015). Instead of working directly with sets of transitions, we label transitions by multiple colored marks, as can be seen in Figure 3.

Let $M = \{0, \ldots, n-1\}$ be a finite set of n contiguous integers called the set of *marks* or *colors*, from now on also written $M = \{0, 1, \ldots\}$ in our examples. We define the set $\mathcal{C}(M)$ of *acceptance formulas* according to the following grammar, where m stands for any mark in M:

$$\alpha ::= \top \mid \bot \mid \mathsf{Inf}(m) \mid \mathsf{Fin}(m) \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

Acceptance formulas are interpreted over subsets of M. For $N \subseteq M$ we define the satisfaction relation $N \models \alpha$ inductively according to the following semantics:

$$N \models \top, \quad N \models \mathsf{Inf}(m) \text{ iff } m \in N, \quad N \models \alpha_1 \wedge \alpha_2 \text{ iff } N \models \alpha_1 \text{ and } N \models \alpha_2,$$
$$N \not\models \bot, \quad N \models \mathsf{Fin}(m) \text{ iff } m \notin N, \quad N \models \alpha_1 \vee \alpha_2 \text{ iff } N \models \alpha_1 \text{ or } N \models \alpha_2.$$

Table 1: Example of acceptance formulas for traditional acceptance conditions.

Büchi	$\mathsf{Inf}(0)$
generalized Büchi	$\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \wedge \mathsf{Inf}(2) \wedge \ldots$
co-Büchi	$\mathsf{Fin}(0)$
Rabin	$(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(3)) \vee \ldots$
Streett	$(\mathsf{Inf}(0) \vee \mathsf{Fin}(1)) \wedge (\mathsf{Inf}(2) \vee \mathsf{Fin}(3)) \wedge \ldots$
parity min even	$\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(3) \wedge \ldots)))$
parity min odd	$\mathsf{Fin}(0) \wedge (\mathsf{Inf}(1) \vee (\mathsf{Fin}(2) \wedge (\mathsf{Inf}(3) \vee \ldots)))$
parity max even	$(((\mathsf{Inf}(0) \wedge \mathsf{Fin}(1)) \vee \mathsf{Inf}(2)) \wedge \mathsf{Fin}(3)) \vee \ldots$
parity max odd	$(((\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge \mathsf{Fin}(2)) \vee \mathsf{Inf}(3)) \wedge \ldots$

An Emerson-Lei automaton is an ω-automaton labeled by marks whose acceptance condition is expressed as a positive Boolean formula on sets of marks that occur infinitely or finitely often in a run. More formally:

Definition 4 (Transition-based Emerson-Lei Automata) A *transition-based Emerson-Lei automaton* (TELA) is a tuple $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ where:

- Q is a finite set of states.
- M is a finite set of marks.
- Σ is a finite input alphabet.
- $\delta \subseteq Q \times \Sigma \times 2^M \times Q$ is a finite set of transitions.
- $q_0 \in Q$ is an initial state.
- $\alpha \in \mathcal{C}(M)$ is an acceptance formula.

Given a transition $d = (q_1, \ell, A, q_2) \in \delta$, we write $d = q_1 \xrightarrow{\ell, A} q_2$; q_2 is said to be a *successor* of q_1, and q_1, the *origin* of d. If $A = \varnothing$, the transition is said to be uncolored, and has no direct impact on the acceptance. A *run* r of $\mathcal{A}$ is an infinite sequence of transitions $r = (s_i \xrightarrow{\ell_i, A_i} s'_i)_{i \geqslant 0}$ in δ^ω such that $s_0 = q_0$ and $\forall i \geqslant 0$, $s'_i = s_{i+1}$. Since δ is finite, for any run r, there exists a position $j_r \geqslant 0$ such that for each $i \geqslant j_r$, the transition $s_i \xrightarrow{\ell_i, A_i} s'_i$ occurs infinitely often in r. Let $\mathsf{Rep}(r) = \bigcup_{i \geqslant j_r} A_i$ be the set of colors **rep**eated infinitely often in r.

A run r is *accepting* if $\mathsf{Rep}(r) \models \alpha$, and we then say that $\mathcal{A}$ accepts the word $(\ell_i)_{i \geqslant 0} \in \Sigma^\omega$. We may then write $r \models \alpha$. The *language* $\mathscr{L}(\mathcal{A})$ is the set of words accepted by $\mathcal{A}$. Two TELA are *equivalent* if they have the same language. By extension, the language of a state $q \in Q$ is the language of the automaton using q as initial state.

Finally, a TELA is *deterministic* if each state has at most one outgoing edge for any given letter, in other words, $\forall (q, v) \in Q \times \Sigma$, $|\{(s, \ell, A, d) \in \delta \mid s = q \wedge \ell = v\}| \leqslant 1$, and it is *complete* if each state has at least one outgoing edge for any given letter, in other words, $\forall (q, v) \in Q \times \Sigma$, $|\{(s, \ell, A, d) \in \delta \mid s = q \wedge \ell = v\}| \geqslant 1$.

The grammar of acceptance formulas can represent many traditional acceptance conditions, as illustrated in Table 1. For parity acceptance, interpreting the colors as numbers amounts to checking whether the minimum or maximum color seen infinitely often is odd or even.

Some subclasses of TELA are of specific interest to us. A *DELA* is a deterministic TELA. If it has parity acceptance, we call it *deterministic parity automaton* (DPA). `ltlsynt` works with *max odd* parity, but the techniques described can be adjusted to other types of parity acceptance. A *NBA* is a TELA with Büchi acceptance, and a *DBA* is the deterministic version.

The following theorem captures some well-established results linking LTL to TELA.

Theorem 2.3 *For any LTL formula there exists an equivalent NBA and an equivalent DPA. There exist some LTL formulas that cannot be represented by an equivalent DBA.*

The following definition and theorem connect the syntactic obligation formula of Section 2.2 to the class of weak TELA:

Definition 5 A *weak automaton* is a TELA in which all transitions that belong to the same strongly connected component have the same color. Without loss of expressivity, one can always recolor weak automata to use a Büchi acceptance condition $\mathsf{Inf}(0)$.

Theorem 2.4 (Černá and Pelánek (2003); Dax et al (2007)) *Any syntactic obligation formula can be converted into an equivalent deterministic weak automaton of minimal size.*

Similarly, the classes of syntactic recurrence and syntactic persistence formula can be connected to specific types of deterministic automata.

Theorem 2.5 (Černá and Pelánek (2003)) *For any syntactic recurrence formula there exists an equivalent deterministic Büchi automaton and dually for any syntactic persistence formula there exists an equivalent deterministic co-Büchi automaton.*

2.4 Operations on Deterministic TELA

We now define operations over deterministic TELA that realize Boolean operations over their language. Let us first extend our definition of acceptance formulas via some syntactic sugar.

Definition 6 (Syntactic sugar for acceptance formulas) Let α be an acceptance formula, then $\neg\alpha$ is the acceptance formula defined inductively by

$$\neg\top = \bot \qquad \neg\mathsf{Inf}(m) = \mathsf{Fin}(m) \qquad \neg(\alpha_1 \wedge \alpha_2) = (\neg\alpha_1) \vee (\neg\alpha_2)$$

$$\neg\bot = \top \qquad \neg\mathsf{Fin}(m) = \mathsf{Inf}(m) \qquad \neg(\alpha_1 \vee \alpha_2) = (\neg\alpha_1) \wedge (\neg\alpha_2)$$

For two acceptance formulas α and β, let us define three additional operations:

$$\begin{aligned}\alpha \to \beta &= \neg\alpha \vee \beta\\ \alpha \leftrightarrow \beta &= (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)\\ \alpha \oplus \beta &= (\alpha \wedge \neg\beta) \vee (\neg\alpha \wedge \beta)\end{aligned}$$

Theorem 2.6 (Complement of a deterministic TELA) *Let $\mathcal{A} = (Q, M, \Sigma, \delta, q_0, \alpha)$ be a deterministic complete TELA. We define* $\mathsf{Complement}(\mathcal{A})$ *as the automaton $(Q, M, \Sigma, \delta, q_0, \neg\alpha)$, and we obviously have $\mathscr{L}(\mathsf{Complement}(\mathcal{A})) = \mathscr{L}(\mathcal{A})^{\complement}$.*

Theorem 2.7 (Generalized product) *Let $\mathcal{A}_1 = (Q_1, M_1, \Sigma, \delta_1, i_1, \alpha_1)$ and $\mathcal{A}_2 = (Q_2, M_2, \Sigma, \delta_2, i_2, \alpha_2)$ be two deterministic TELA over the same alphabet using two disjoint sets of colors M_1 and M_2.*

For any $\odot \in \{\wedge, \vee, \to, \leftrightarrow, \oplus\}$ let $\mathsf{Product}_{\odot}(\mathcal{A}_1, \mathcal{A}_2)$ *be the TELA $(Q, M, \Sigma, \delta, i, \alpha)$ where:*

- $Q = Q_1 \times Q_2$,
- $\delta = \{((s_1, s_2), \ell_1, m_1 \cup m_2, (d_1, d_2)) \mid (s_1, \ell_1, m_1, d_1) \in \delta_1,\ (s_2, \ell_2, m_2, d_2) \in \delta_2,\ \ell_1 = \ell_2\}$,
- $i = (i_1, i_2)$,
- $\alpha = \alpha_1 \odot \alpha_2$

Then, it follows that:

$$\begin{aligned}\mathscr{L}(\mathsf{Product}_{\wedge}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathscr{L}(\mathcal{A}_1) \cap \mathscr{L}(\mathcal{A}_2)\\ \mathscr{L}(\mathsf{Product}_{\vee}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathscr{L}(\mathcal{A}_1) \cup \mathscr{L}(\mathcal{A}_2)\\ \mathscr{L}(\mathsf{Product}_{\to}(\mathcal{A}_1, \mathcal{A}_2)) &= \mathscr{L}(\mathcal{A}_1)^{\complement} \cup \mathscr{L}(\mathcal{A}_2)\\ \mathscr{L}(\mathsf{Product}_{\leftrightarrow}(\mathcal{A}_1, \mathcal{A}_2)) &= (\mathscr{L}(\mathcal{A}_1) \cap \mathscr{L}(\mathcal{A}_2)) \cup (\mathscr{L}(\mathcal{A}_1)^{\complement} \cap \mathscr{L}(\mathcal{A}_2))^{\complement}\\ \mathscr{L}(\mathsf{Product}_{\oplus}(\mathcal{A}_1, \mathcal{A}_2)) &= (\mathscr{L}(\mathcal{A}_1) \setminus \mathscr{L}(\mathcal{A}_2)) \cup (\mathscr{L}(\mathcal{A}_2) \setminus \mathscr{L}(\mathcal{A}_1))\end{aligned}$$

The above product can be adapted to the case where the set of colors M_1 and M_2 used by each automaton are not disjoint, by first renumbering the colors of one automaton to make sure they are unique. In the end, the number of colors used by the above product is $|M_1| + |M_2|$.

In the context of reactive synthesis, we will consider an alphabet $\Sigma = \mathbb{B}^{I \cup O}$. The edges of a TELA are labeled by input and output propositions.

2.5 Parity Games

We view parity games as specialized versions of TELA where each state is owned by one of two players.

Definition 7 (Parity Game) A *parity game* is a TELA of the form $\mathcal{G} = (Q_0 \uplus Q_1, M, \Sigma, \delta, q_0, \alpha)$ such that:

- Q_0 and Q_1 are two *disjoint* sets of states respectively controlled by player 0 ("the environment") and player 1 ("the controller").
- α is a parity max odd condition.

A run of $\mathcal{G}$ is called a *play*; it is said to be *winning* for player 1 if it is an accepting run, otherwise, if the run is non-accepting, it is winning for player 0.

As shown in Figure 3, states in Q_0 will be denoted $\bigcirc$, and states in Q_1, $\diamond$. Intuitively, player 0 and player 1 take turns and pick a successor in accordance to the current state and the transitions defined in δ. Player 1 tries to ensure that the resulting run is accepting, while player 0 is actively preventing this outcome.

A *memoryless strategy* for a player X (with $X \in \{0,1\}$) is a function $\sigma_X : Q_X \to \delta$ such that $\sigma_X(q)$ is always an outgoing transition of $q \in Q_X$.

A play $p = (s_i \xrightarrow{\ell_i, A_i} s'_i)_{i \geqslant 0}$ is said to be *consistent* with σ_X if $\forall s_i \in Q_X$, $(s_i \xrightarrow{\ell_i, A_i} s'_i) = \sigma_X(s_i)$, that is, the transition leaving any state in Q_X in that run is determined by σ_X. Finally, σ_X is said to be a *winning strategy* for player X if any play that is starting in q_0 and consistent with σ_X is winning for player X.

Parity games are known to be *positionally determined*, this means that one of the two players has a memoryless winning strategy. Büchi games are *positionally determined* too, however as seen in Theorem 2.3, DBA are less expressive than DPA; we therefore focus on DPA. Several algorithms exist for finding winning strategies in parity games (Zielonka, 1998; Jurdziński, 2000; van Dijk, 2018), and those that are defined with state-based acceptance can easily be adapted to transition-based acceptance. Our implementation is based on Zielonka's algorithm with improvements taken from van Dijk (2018).

2.6 Mealy Machines

Reactive controllers produce for an input stream of Boolean valuations a matching output stream. We model this behavior thanks to a common finite state model known as *Mealy machines*, as shown in Figure 5. In particular, we will use Mealy machines to represent the winning strategies we obtain while solving parity games. However, a given specification may yield multiple compatible output valuations for a given input: our model must therefore account for this peculiarity of reactive synthesis.

Definition 8 An *Incompletely specified Generalized Mealy Machine* (IGMM) is a tuple $M = (I, O, Q, q_{init}, \delta, \lambda)$, where I is a set of *input propositions*, O a set of *output propositions*, Q a finite set of *states*, q_{init} an *initial state*, $\delta\colon \left(Q, \mathbb{B}^I\right) \to Q$ a partial *transition function*, and $\lambda\colon \left(Q, \mathbb{B}^I\right) \to 2^{\mathbb{B}^O} \setminus \{\varnothing\}$ an *output function* such that $\lambda(q,i) = \top$ when $\delta(q,i)$ is undefined. If δ is a total function, we then say that M is *input-complete*.

It is worth noting that the transition function is *input-deterministic* but not complete with regards to Q as $\delta(q, i)$ could be undefined. (When $\delta(q, i)$ is undefined, the Mealy machine will be free to do anything, hence the convention that $\lambda(q, i) = \top$.) Furthermore, the output function may return a set of valuations for a given input valuation and state. This is not an unexpected definition from a reactive synthesis point of view, as discussed earlier.

Definition 9 (Semantics of IGMMs) Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ be an IGMM. For all $u \in \mathbb{B}^I$ and $q \in Q$, if $\delta(q, u)$ is defined, we write that $q \xrightarrow{u/v} \delta(q, u)$ for all $v \in \lambda(q, u)$. Given two infinite sequences of valuations $\iota = i_0 \cdot i_1 \cdot i_2 \cdots \in (\mathbb{B}^I)^\omega$ and $o = o_0 \cdot o_1 \cdot o_2 \cdots \in (\mathbb{B}^O)^\omega$, we note $(\iota, o) \Vdash M_q$ if and only if:

- either there is an infinite sequence of states $(q_j)_{j \geqslant 0} \in Q^\omega$ such that $q = q_0$ and $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} q_2 \xrightarrow{i_2/o_2} \cdots$;
- or there is a finite sequence of states $(q_j)_{0 \leqslant j \leqslant k} \in Q^{k+1}$ such that $q = q_0$, $\delta(q_k, i_k)$ is undefined, and $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} \cdots q_k$.

We then say that starting from state q, M *produces* output o given the input ι.

Note that if $\delta(q_k, i_k)$ is undefined, the machine is allowed to produce an arbitrary output from then on. Furthermore, given an input word ι, there is always at least one output word o such that $(\iota, o) \Vdash M_q$ but there might be more. Eventually, when that machine is encoded into a circuit, we will have to settle on a single output word per input word, but in the meantime, this additional flexibility can be used for simplification.

Definition 10 (Realizability of an LTL formula by an IGMM) Let I and O be two disjoint sets of input and output propositions. Given two sequences $\iota = i_0 \cdot i_1 \cdot i_2 \cdots \in (\mathbb{B}^I)^\omega$ and $o = o_0 \cdot o_1 \cdot o_2 \cdots \in (\mathbb{B}^O)^\omega$, we denote by $\iota \wedge o$ the sequence $(i_0 \wedge o_0) \cdot (i_1 \wedge o_1) \cdot (i_2 \wedge o_2) \cdots \in \mathbb{B}^{O \cup I}$.

Let φ be an LTL formula built upon $I \cup O$. We say that an IGMM $M = (I, O, Q, q_{init}, \delta, \lambda)$ *realizes* the specification φ if for any pair of sequences $\iota \in (\mathbb{B}^I)^\omega$ and $o \in (\mathbb{B}^O)^\omega$, we have

$$(\iota, o) \Vdash M_{q_{init}} \implies \iota \wedge o \vDash \varphi$$

We say that a specification φ is *realizable* if there exists an IGMM that realizes it.

In other words, the machine M *realizes* the specification φ if, regardless of the provided input ι, any output sequence o that M may produce will (together with ι) satisfy φ.

Definition 11 (Variation and specialization) Let $M = (I, O, Q, q_{init}, \delta, \lambda)$ and $M' = (I, O, Q', q'_{init}, \delta', \lambda')$ be two IGMMs. Given two states $q \in Q$, $q' \in Q'$, we say that q' is a *variation* of q if $\forall \iota \in (\mathbb{B}^I)^\omega, \{o \mid (\iota, o) \Vdash M'_{q'}\} \cap \{o \mid (\iota, o) \Vdash M_q\} \neq \varnothing$; and q' is a *specialization* of q if $\forall \iota \in (\mathbb{B}^I)^\omega, \{o \mid (\iota, o) \Vdash M'_{q'}\} \subseteq \{o \mid (\iota, o) \Vdash M_q\}$. We

say that M' is a variation (resp. specialization) of M if q'_{init} is a variation (resp. specialization) of q_{init}.

Intuitively, all the input-output pairs accepted by a specialization q' in M' are also accepted by q in M. Therefore, if M_q realizes a specification φ, then its specialization $M'_{q'}$ also realizes φ. Figure 6 shows a specialization of Figure 5. This specialization relation can obviously be used to compute smaller machines that still comply with a given specification.

Finally, in order for two states to be a variation of one another, for all possible inputs they must be able to agree on a common output behavior. This property can be used to reduce the number of states in a given machine; however, additional care has to be taken as we will show in Section 4.

2.7 The LTL Reactive Synthesis Problem

The LTL Reactive Synthesis problem can be formulated as follow: given an LTL formula φ over a set of variable $O \uplus I$ partitioned as input and output, does there exist an IGMM M that realizes φ?

This is the problem posed by SYNTCOMP (Jacobs et al, 2017, 2019), the annual Reactive synthesis competition, which distinguishes two subproblems:

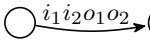
- in the *realizability* track, the goal is simply to decide the above question without providing M;
- in the *synthesis* track, the goal is to construct a reactive controller that realizes the specification, or to state that the specification is not realizable.

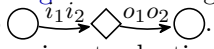
In the latter track, the controller has to be constructed as an AIG circuit expressed in the AIGER format (Biere, 2007). Section 5 will be devoted to the transformation of IGMM into AIG circuits.

3 Translating LTL Formulas into Games

3.1 Different Approaches

Figure 8 shows four approaches for transforming an LTL formula into a parity game. In `ltlsynt`, these approaches can be selected using the `--algo` option.

The `ds` branch corresponds to the straightforward approach described in the introduction. The formula is first converted into a non-deterministic Büchi automaton using Spot's standard translation (Duret-Lutz, 2014). Then it is determinized into a parity automaton with a variant of Safra's determinization (Redziejowski, 2012). The result, which may look like Fig. 3 is then converted into a game as in Fig. 4 by splitting transitions of the form 

into 

. By ensuring that at most one intermediate state is created per input valuation, determinism is preserved (as discussed in Section 3.3). The game is also made input-complete, to ensure that the environment can freely chose among all input valuations.

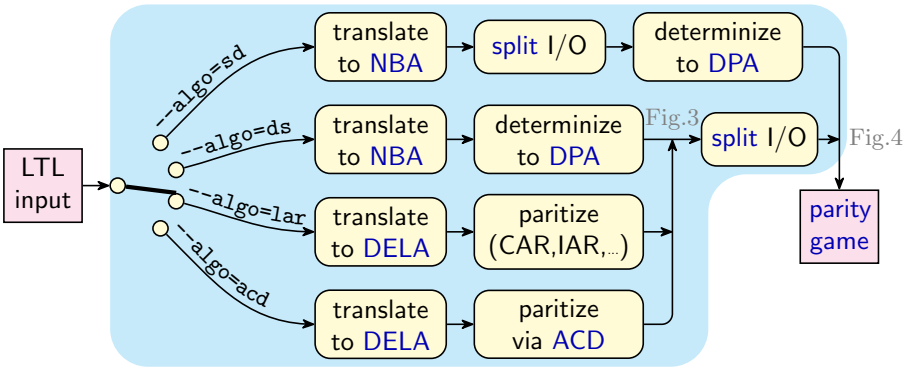


Fig. 8: Zoom on the block "*translate φ_i to game*" from Fig. 7. The `--algo` option of `ltlsynt` will select one of several ways to construct a parity game.

The `sd` approach is similar, except that the split is performed before the determinization. This slightly counter-intuitive order is motivated by the fact that at a given time, the determinization then has to deal only with valuations in 2^O or valuations in 2^I, while in the previous approach, it had to cope with $2^{O \cup I}$ valuations.

The other two approaches are attempts to reduce the costs of obtaining a DPA. This is done by first converting the formula into a deterministic TELA (*DELA*), i.e., using arbitrary acceptance condition, and then converting this automaton into a DPA. The conversion to deterministic TELA is described in the next section. The two approaches differ in how the *paritization* (i.e., the conversion of the TELA to a parity automaton) is done.

The `lar` approach uses a paritization procedure based on *latest appearance records* with many improvements described in previous work (Renkin et al, 2020).

The newest `acd` approach replaces the above paritization by one based on the *Alternating Cycle Decomposition* (ACD) (Casares et al, 2021, 2022). It is guaranteed to produce automata that are at most as big as `lar`.

3.2 From LTL to Deterministic TELA

There exist a number of tools, such as `delag` (Müller and Sickert, 2017) or `ltl3tela` (Major et al, 2019) for transforming LTL formulas into deterministic TELA (DELA). Spot's own built-in procedure for this purpose, used by `ltlsynt`, is inspired from `delag`.

Algorithm 1 shows a slightly simplified view of how Spot translates φ into a deterministic TELA. Formula φ is assumed to be in a negative normal form (i.e., $\neg$ have been pushed down in front of the atomic propositions, and implications have been rewritten away, as in Theorem 2.1) where equivalence ($\leftrightarrow$) and xor ($\oplus$) can still be used.

It builds upon the following procedures:

Algorithm 1 Translation of an LTL formula into a deterministic EL-automaton.

Input: An LTL formula φ in NNF, and an optional binary operator op.
Output: A deterministic EL-automaton.
function ToDELA($\varphi, op = \bot$)
  if φ matches $\underbrace{\mathsf{X}\mathsf{X}\ldots\mathsf{X}}_{i \text{ copies of } \mathsf{X}}\ \alpha$ **then**
    $\mathcal{A}_\alpha \leftarrow$ ToDELA(α)
    create $\mathcal{A}_\varphi$ from $\mathcal{A}_\alpha$ by prepending i states in the obvious way
    return $\mathcal{A}_\varphi$
  if φ matches $f_1 \odot \cdots \odot f_n$ for $\odot \in \{\wedge, \vee, \leftrightarrow, \oplus\} \backslash \{op\}$ and $n \geqslant 2$ **then**
    Partition $\{f_1, ..., f_n\}$ as $S \cup O \cup R$ with $\begin{cases} S\text{: } \textit{suspendable formulas} \\ O\text{: } \textit{obligation formulas} \\ R\text{: anything else} \end{cases}$
    $\mathcal{A} \leftarrow$ Product$_\odot$(ToDELA($\bigodot_{f \in R} f, \odot$), BuildMinWDBA($\bigodot_{f \in O} f$))
    for $s \in S$ **do**
      $\mathcal{A} \leftarrow$ ProductSusp$_\odot$($\mathcal{A}$, ToDELA($s, \odot$))
    return $\mathcal{A}$
  if φ is a syntactic obligation **then**
    return BuildMinWDBA(φ)
  if φ matches $\mathsf{G}(\bigwedge_i \mathsf{F}\alpha_i)$ where α_i are syntactic guarantees **then**
    return GFGuaranteeToDBA(φ)
  if φ matches $\mathsf{F}(\bigvee_i \mathsf{G}\alpha_i)$ where α_i are syntactic safeties **then**
    return Complement(GFGuaranteeToDBA($\neg\varphi$))
  return Determinize(ToNBA(φ))

Product$_\odot$ builds a product of two deterministic automata using a standard synchronous product, and combines their acceptance conditions using $\odot \in \{\wedge, \vee, \leftrightarrow, \oplus\}$ (see Theorem 2.7).

ProductSusp$_\odot$ also builds a product, but assumes the second argument is a suspendable property. As a consequence, the actual product needs only to be performed in the accepting SCCs of the first automaton. This construction is similar to constructions discussed by Müller and Sickert (2017) and Babiak et al (2013), and is justified by Theorem 2.2.

BuildMinWDBA transforms any obligation formula into a minimal weak deterministic Büchi automaton using a procedure described by Dax et al (2007).

GFGuaranteeToDBA is an algorithm inspired from a similar one by Esparza et al (2018) and discussed in more details later. It can convert any formula of the form $\mathsf{G}(\bigwedge_i \mathsf{F}\alpha_i)$, where α_is are syntactic guarantee formulas, into a deterministic Büchi automaton.

Complement dualizes the acceptance condition of any deterministic TELA to complement it (see Theorem 2.6).

ToNBA converts an LTL formula into a non-deterministic Büchi automaton. (Duret-Lutz, 2014)

Determinize determinizes a non-deterministic automaton into a parity automaton, using a Safra-based algorithm. Spot implements Redziejowski's algorithm (2012).

Let us explain this algorithm by starting on line 20, which is the only necessary line. It is well known that (1) any LTL formula can be converted into a non-deterministic Büchi automaton with an exponential blowup, (2) not all LTL formulas may be represented as deterministic Büchi automaton. As a consequence, applications that require deterministic automata usually rely on more complex acceptance conditions. For instance there exists procedures for transforming non-deterministic Büchi automata (NBA) into deterministic Rabin automata or into deterministic parity automata (DPA), and those also have exponential blowups. Line 20 uses such a determinization to obtain a DPA from an LTL translation, at the cost of a doubly-exponential construction in the worst case (Kupferman and Rosenberg, 2010).

The other lines are therefore here to help reduce the cost of this construction, by providing specialized constructions for certain subclasses of formulas, and by decomposing the formula into smaller parts that are cheaper to translate and later recompose.

The idea behind the decomposition is that if a formula φ has the shape $f_1 \odot \cdots \odot f_n$, for some Boolean operator $\odot \in \{\vee, \wedge, \leftrightarrow, \oplus\}$, then the deterministic TELA $\mathcal{A}_\varphi$ can be obtained by making a synchronous product of the deterministic TELA A_{f_i} obtained for each f_i, using operator $\odot$ to combine the acceptance conditions. In this process, the translation of the smaller f_i might use more specialized algorithms. If more specialized algorithms are not available, empirical experiences tell us that it is usually best to translate φ without decomposing it, because the complexity of the translation is mostly proportional to the size of the produced automata anyway.

It is important to be able to decompose on $\leftrightarrow$ and $\oplus$ and not just $\wedge$ and $\vee$. The reason is that $\leftrightarrow$ in particular often occurs in synthesis specifications, and translating $f \leftrightarrow g$ as $\mathsf{Product}_\odot(\mathcal{A}_f, \mathcal{A}_g)$ involves only one product, while the equivalent formula $(f \wedge g) \vee (\neg f \wedge \neg g)$ would require three products. The same reduction in number of products is achieved for $\oplus$, but would not be obtained for $\rightarrow$. Hence we are happy to assume implications have been removed when the input formula was rewritten in negative normal form.

This idea of decomposing the LTL formula to call specialized constructions is similar to that used in the tool `delag` (Müller and Sickert, 2017). In `delag`, a formula is split into syntactic safety, syntactic guarantee, another class called *fairness*, and anything else. We generalize this slightly by using syntactic obligations (a super-class of syntactic safety and syntactic guarantee), suspendable formulas (a super-class of `delag`'s *fairness* class), and anything else. The partition into these three classes is done on lines 8–9.

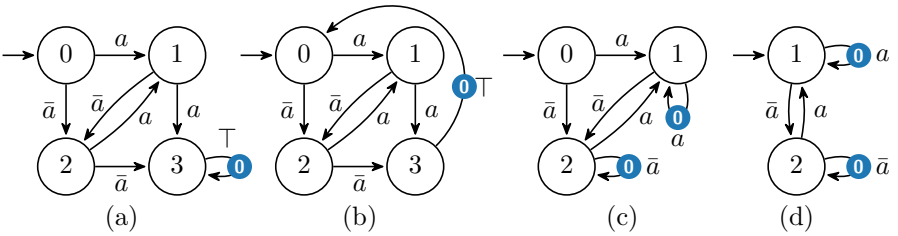


Fig. 9: (a) a DBA for $\mathsf{F}(a \leftrightarrow \mathsf{X}a)$; (b,c,d) three DBAs for $\mathsf{GF}(a \leftrightarrow \mathsf{X}a)$.

Several operations occur on line 10. Then, all formulas of set R, which do not belong to a subclass for which we have a dedicated construction, are translated recursively with $\mathsf{ToDELA}(\bigodot_{f\in R} f, \odot)$. The reason for the recursion, as opposed to calling $\mathsf{Determinize}(\mathsf{ToNBA}(\bigodot_{f\in R} f))$ directly, is that if R contains a single formula whose top-level operator is a Boolean operator different from $\odot$, it might be decomposed again.

Similarly, all formulas of set O, are translated into a minimal weak deterministic Büchi automaton following a technique of Dax et al (2007); essentially, for a syntactic obligation formula α, the following steps, which are embodied in the BuildMinWDBA function, produce a minimal WDBA (Theorem 2.4):

1. translate α into an NBA N_α,
2. ignoring accepting states, use the powerset construction on N_α to obtain the deterministic structure D_α,
3. any SCC of D_α that intersects an accepting SCC of N_α in the synchronous product of $N_\alpha \otimes D_\alpha$ should be marked as accepting
4. now D_α is a weak deterministic automaton for α, and it can be minimized using Löding's algorithm (2001).

The automata resulting from the translations of $\bigodot_{f\in R} f$ and $\bigodot_{f\in O} f$ are then composed by taking their product (we are still on line 10). (For simplicity we assume here that if O or R is empty, its translation gives a simple universal automaton. But the implementation of course skips the product in this case.) Spot's implementation of $\mathsf{Product}_\odot$ has an extra trick, used here: when one of the operands is a weak automaton, it does not need to contribute colors to the resulting product.

Lines 11–12 integrate the suspendable formulas to the result one after the other, by translating them recursively, and then using the $\mathsf{ProductSusp}_\odot$ method.

When the ToDELA function is called, the input formula has generally been simplified using folklore rewritings such a changing $(\mathsf{X}f_1) \odot (\mathsf{X}f_2)$ into $\mathsf{X}(f_1 \odot f_2)$. For this reason, lines 4–7 strip any leading Xs in order to apply the construction to the rest of the formula.

Finally, lines 14–19 detect subclasses of formulas for which a specialized construction exists. If φ is a syntactic obligation, the aforementioned BuildMinWDBA function is used.

If formula φ has the form $\mathsf{G}(\bigwedge_i \mathsf{F}\alpha_i)$, and α_i is a syntactic guarantee formula, then the following algorithm from Esparza et al (2018) can be used:

1. Translate $\bigwedge_i \mathsf{F}\alpha_i$ into a DBA A with a single accepting state that is terminal. The simple structure of $\bigwedge_i \mathsf{F}\alpha_i$ makes this possible. (Fig. 9(a) gives an example.)
2. Build A_φ from A by redirecting the loop of the accepting states to the initial state. (Fig. 9(b).)

Our implementation of GFGuaranteeToDBA improves upon the above by redirecting not the outgoing transition of the accepting states, but its incoming transitions (Fig. 9(c)): for each transition going to the terminal state, pretend that this letter is read from the initial state to pick the new destination. If such a transition is necessarily preceded by some forced sequence of valuations, any suffix of this sequence of valuations can actually be replayed from the initial state, allowing tighter loops. Finally, note that state 0 and 1 of Fig. 9(c) have the same successors, so they can be merged to obtain the automaton Fig. 9(d).

3.3 Splitting Automata

Once a parity automaton has been obtained (as in Fig. 3), we split its transitions to separate input signals from output signals, and obtain a parity game (as in Fig. 4). In this game player 1 ("the controller") selects the output valuations and can effectively react to the input valuations selected by player 0 ("the environment").

On the `--algo=sd` path of Fig. 8, we apply this split operation to non-deterministic Büchi automata. Therefore, below, we define it for any acceptance condition, and partition the states of the resulting automaton as $Q_0 \uplus Q_1$, with $Q_{0/1}$ being the set of states belonging to player 0/1, allowing this automaton to be interpreted as a game if desired.

Definition 12 (Splitting an Emerson-Lei Automata) Let $\mathcal{A} = (Q, M, \mathbb{B}^{I \uplus O}, \delta, q_0, \alpha)$ be a TELA. The *split* of $\mathcal{A}$ is the TELA $\mathcal{A}_s = (Q_0 \uplus Q_1, M, \mathbb{B}^I \cup \mathbb{B}^O, \delta_0 \cup \delta_1, q_0, \alpha)$, where:

- $Q_0 = Q$,
- $Q_1 = Q \times \mathbb{B}^I$,
- $\delta_0 = \left\{ s \xrightarrow{\ell_i, \varnothing} (s, \ell_i) \;\middle|\; \ell_i \in \mathbb{B}^I, \ell_o \in \mathbb{B}^O, s \xrightarrow{\ell, A} d \in \delta, \ell = \ell_i \wedge \ell_o \right\}$,
- $\delta_1 = \left\{ (s, \ell_i) \xrightarrow{\ell_o, A} d \;\middle|\; \ell_i \in \mathbb{B}^I, \ell_o \in \mathbb{B}^O, s \xrightarrow{\ell, A} d \in \delta, \ell = \ell_i \wedge \ell_o \right\}$.

Figure 10 shows a small example demonstrating how edges with identical inputs are fused into a single edge for the environment player. This ensures that states in Q_0 are always deterministic, while states in Q_1 are only deterministic if the input TELA is deterministic.

In a similar manner to transitions, we can split runs: if $r \in (\mathbb{B}^{I \uplus O})^\omega$, $r = (r_i)_{i \geqslant 0}$, and we consider $\forall i \geqslant 0$, $r_i = r_i^I r_i^O$ where $r_i^I \in \mathbb{B}^I$ and $r_i^O \in \mathbb{B}^O$

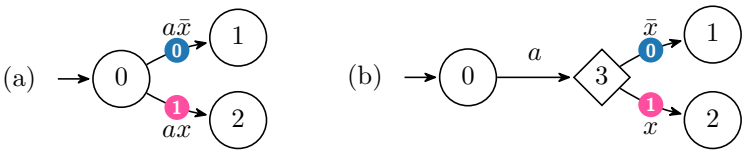


Fig. 10: (a) Original automaton with $I = \{a\}$ and $O = \{x\}$. (b) Split automaton interpreted as a game with Q_0 as round states, and Q_1 as diamonds states.

are again the respective projections of r_i. Then the split of r is the run $r_s = r_0^I r_0^O r_1^I r_1^O \dots$ in $(\mathbb{B}^I \cdot \mathbb{B}^O)^\omega$. Obviously, $\mathcal{A}$ accepts a run r if and only if $\mathcal{A}_s$ accepts r_s: splitting a TELA preserves the underlying specification as a game.

Two Implementation Details

The definition of TELA and game used here are deliberately simplified concepts to ease the notations. In particular, our implementation of TELA labels transitions with subsets of valuations (i.e., arbitrary Boolean formulas over $I \cup O$) instead of only one valuation at a time, in order to reduce the number of transitions stored and iterated upon. The splitting procedure detailed above is easily adapted to this setting: using subsets of valuations as ℓ_i in (s, ℓ_i) will also help reduce the size of Q_1.

Furthermore, if the acceptance condition is *parity max odd*, the transitions introduced in δ_0 can be colored using the minimal color used in the outgoing transitions of their destination states (with no acceptance mark being considered smaller than any color). For instance if the automaton of Fig 10(b) has *parity max odd* acceptance, the transition between states 0 and 3 can be colored with 0, the minimum color seen in the outgoing transitions of 3.

4 Simplifying Winning Strategies using Generalized Mealy Machines

Once a winning strategy for player 1 has been found for a game, we extract it as an Incompletely specified Generalized Mealy Machine. This is done in a simple way: first every outgoing edge of a state in Q_1 that is not part of the strategy is removed, as well as all colors. Then the reachable part of the remaining automaton can be seen as a collection of pairs of transitions of the form $\bigcirc \xrightarrow{i_1 i_2} \diamond \xrightarrow{o_1 o_2} \bigcirc$ which we fuse back into $\bigcirc \xrightarrow{i_1 i_2 / o_1 o_2} \bigcirc$ to obtain a Mealy machine. For instance the Mealy machine in Fig. 5 is obtained from the game together with the strategy highlighted by the thick transitions in Fig. 4 using this approach.

These Mealy machines will then be encoded into And-Inverter Graphs as we discuss in Section 5, but first they can be simplified.

If a Mealy machine has n states, the AIG will use $\lceil \log n \rceil$ latches to remember its current state; and reducing this number will usually reduce the size

of the AIG (but that is not always true). For instance the Mealy machine in Fig. 5 requires two latches (two bits are needed to distinguish the three states), while the one of Fig. 6 requires only one latch (as shown in Fig. 2).

As shown in Fig. 4, a strategy determines for each state which transition to choose. As transitions are labeled by arbitrary sets of valuations (over the output propositions), the strategy implicitly defines a set of valid output valuations to respond to the last input valuation. The controller is free to pick any one of these valuations and is guaranteed to respect the initial specification.

This brings us back to the notion of specialization of Mealy machines, introduced in Section 2.6. Please recall that intuitively, a Mealy machine $\mathcal{M}_1$ is a specialization of a Mealy machine $\mathcal{M}_2$ if for any input sequence, any associated output produced by $\mathcal{M}_1$ can also be produced by $\mathcal{M}_2$.

Our goal is to derive a specialization of a given Mealy machine having fewer states. In the following subsections, we give an overview of two ways to perform such a reduction: in the first one, called *minimization*, we seek to obtain the specialization having the fewest states possible. In the second one, called *reduction*, we seek to reduce the number of states of the machine, without necessarily achieving the optimal result. The later reduction is motivated by the high computational complexity of the minimization: the problem is known to be NP-complete (Pfleeger, 1973).

These two simplifications are further detailed in a previous work (Renkin et al, 2022), so we only provide intuitions here.

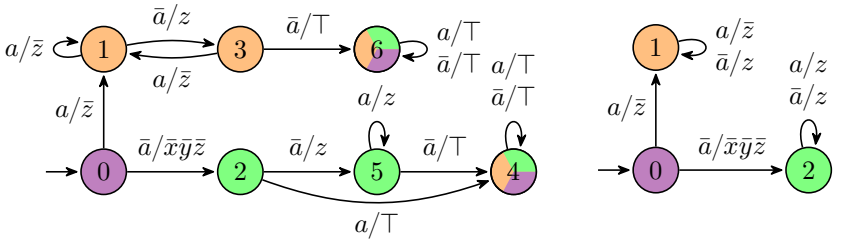
4.1 SAT-Based Minimization of IGMM

Here we discuss a minimization procedure based on the approach presented by Abel and Reineke (2015). Besides improving their algorithm on several points, we generalize this approach to our strictly more expressive Mealy machine model IGMM.

If the Mealy machine was completely specified, we would minimize it by computing a quotient of equivalent states. However, the freedom we get from the incomplete specification allows some of the states to belong to several classes of that equivalence relation, depending on how we decide to specialize those states. To capture this notion of group of compatible states, we define variation classes, and require three additional properties: variation classes have to cover the entire automaton, they have to behave similarly with respect to their successor classes and their possible output.

Definition 13 (Variation class) Given an IGMM $M = (I, O, Q, q_{\text{init}}, \delta, \lambda)$, a *variation class* $C \subseteq Q$ is a set of states, such that all elements are pair-wise variations (See Def. 11). That is $\forall q, q' \in C, q'$ is a variation of q. In the remainder of this section, we will call a variation class simply class as there is no ambiguity.

Note that variation classes should not be mistaken for equivalence classes, since being a variation is not an equivalence relation.



(a) Original IGMM M (b) Minimal specialization of M

Fig. 11: Minimization example

Definition 14 (Cover condition) We say that a set of classes S covers the machine M if every state of M appears in at least one class.

Definition 15 (Closure condition) We say a set of classes S is closed if for all $C_j \in S$ and for all input $i \in \mathbb{B}^I$ there exists a class $C_k \in S$ such that for every state q of C_j we have either $\delta(q, i) \in C_k$ or $\delta(q, i)$ is undefined.

Definition 16 (Nonemptiness condition) We say that a class C has a nonempty output if $\bigcap_{q \in C} \lambda(q, i)$ is not empty for all input $i \in \mathbb{B}^I$.

A set of classes that satisfies the three previously described conditions gives rise to a specialization of the original machine. Therefore, finding the minimal number of classes able to satisfy the conditions amounts to finding the minimal specialization.

For instance in Figure 11a, colors are used to represent three variation classes that satisfy the three additional constraints. (States 4 and 6 belong to the three classes.) The minimal corresponding IGMM is the one of Figure 11b.

In order to find such a decomposition, we fix the number of classes n and encode the conditions given above as a SAT problem. Once we have found the smallest n for which a satisfying assignment has been found by a SAT solver, we can extract the minimal IGMM from it.

A basic iterative algorithm would be to ask if the conditions can be satisfied with one class, and if not, increase the number of classes by one. This naive approach can be improved as shown in Algorithm 2 by first computing a lower bound on the number of classes necessary: for each pair of states q and q', we test if they are variations of one another. If not, these two states cannot be in the same class. We therefore compute a set of states such that no two states in the set are variations of one another. The size of this set gives us a lower bound on the number of classes necessary.

Algorithm 2 SAT-based minimization

```
bool[][] mat ← isNotVariationOf(M)   ▷ Computing the variation matrix
set P ← extractPartialSol(mat)       ▷ Looking for a partial solution P
clauses ← empty list
for n ← |P| to |Q| − 1 do            ▷ Using the lower bound inferred from P
    addCoverCondition(clauses, M, P, mat, n)
    addClosureCondition(clauses, M, P, mat, n)
    (sat, solution) ← satSolver(clauses)   ▷ Solving cover & closure cond.
    while sat do
        if verifyNonEmpty(M, solution) then
            return buildMachine(M, solution)
        addNonemptinessCondition(clauses, M, solution)
        (sat, solution) ← satSolver(clauses)
return copyMachine(M)                ▷ If no solution has been found, return M
```

4.2 Bisimulation-based Reductions

Our second approach relies on an adaptation of a procedure called bisimulation (Babiak et al, 2013) used to reduce the size of an ω-automaton. Again, describing the method in detail is beyond the scope of this paper because this work has been previously published (Renkin et al, 2022); however, we would like to share some insights about how this transformation works. This procedure uses as an intermediate step an association between a state of the automaton and a Boolean formula called *signature*. This signature is computed iteratively in such a way that two states with the same signature recognize the same language. We will now present two algorithms based on an adaptation of this signature-based approach that seeks to reduce an IGMM.

4.2.1 Bisimulation Reduction

Definition 17 (Bisimilarity) Two states q and q' are *bisimilar* if q is a specialization of q' and q' is a specialization of q.

For instance states 4 and 6 in Figure 11a are bisimilar. We can therefore replace the edge that goes from 3 to 6 by an edge that goes from 3 to 4, thereby obtaining a machine with 6 states.

Generally, the *bisimulation reduction* of an automaton is the quotient of that automaton with respect to the above bisimilarity relation. However, this reduction does not take any advantage of the flexibility provided by an IGMM.

4.2.2 Bisimulation Reduction with Output Assignment

In order to merge more states, we introduce an additional preprocessing step called "output assignment." Consider that we have two states q and q' such that for every input sequence, any output that can be associated to a run starting at q can also be produced when starting at q'. If q and q' are not

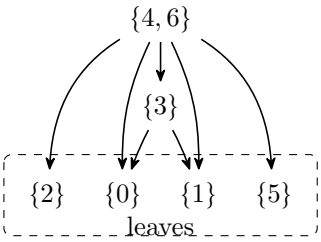


Fig. 12: Specialization graph of the IGMM of Fig. 11a

q		$r(q)$
0	→	0
1	→	1
2	→	2
3	→	1
4	→	1
5	→	5
6	→	1

Fig. 13: Chosen representative mapping.

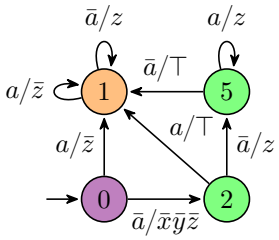


Fig. 14: IGMM obtained by reducing that of Fig. 11a

bisimilar, it means that a run starting at q' can produce a larger set of output sequences than a run starting in q for the same input sequence. By restricting what can be produced from q' to match what is produced from q, we have effectively made them bisimilar and they can therefore be merged using the approach described above. Note that as we restrict the outputs from q', the resulting machine is a specialization of the original one.

To achieve this, we introduce a specialization relation based on the signature. This relation shown in Figure 12 gives us a relation of specialization between states. In this example, state 0 is a specialization of states 3, 4 and 6. The main idea is to associate to each state a state that specializes it. It gives us a representative function given in Figure 13. Once this representative function is found we restrict the output of each state to match the output given by the representative function (i.e., $\lambda(q, i) = \lambda(r(q), i)$). The resulting specialization of the original IGMM can then be reduced by bisimulation (Fig. 14). Note that this procedure produces a machine with as many states as leaves in the specialization tree.

5 Encoding Generalized Mealy Machines as And-Inverter Graphs

The final step in our pipeline is to encode the obtained Mealy machine as an And-Inverter Graph (AIG). And-Inverter Graphs are a special type of directed acyclic graph which is widely used in logic synthesis (Mishchenko et al, 2006; Brayton and Mishchenko, 2010) either as the final circuit or as an intermediate representation before further optimizations. The most common format to represent such circuits is called *AIGER* (Biere, 2007), which is also the output format used in the synthesis-tracks of SYNTCOMP (Jacobs et al, 2017, 2019).

As previously mentioned, AIGs correspond to circuits built from *and gates* with exactly two inputs ⊐, *negations* ○ which can appear on edges and negate the signal if present, and *latches* □ that delay their input by one tick, and output 0 initially.

The input corresponds to an Incompletely specified Generalized Mealy Machine which we want to encode as AIG. Note that the actual behavior of the

AIG only needs to correspond to a specialization of the input machine. Recall that the output function for IGMMs is defined as $\lambda\colon (Q, \mathbb{B}^I) \to 2^{\mathbb{B}^O} \setminus \{\varnothing\}$. This means that for a given state and input valuation $\mathbb{B}^I$ there possibly exist multiple output valuations in $\mathbb{B}^O$ compatible with the specification. Such non-determinism can however not be expressed by a logical circuit, and we therefore need to choose one of the possible valuations, causing the encoded machine to be a specialization of the input machine. We use a function called ChooseOneValuation to resolve these choices when we do not have further constraints. This function takes an output label $L \in 2^{\mathbb{B}^O} \setminus \{\varnothing\}$ and returns a valuation $o \in \mathbb{B}^O$ such that $o \in L$. (Our implementation attempts to select some o that maximizes the number of do-not-care variables and then maximize the number of variables that are set to $\bot$.)

5.1 Encoding Using Boolean Functions

To encode the machine $M = (I, O, Q, q_{init}, \delta, \lambda)$ as AIG, we assume, without loss of generality, that each state is associated to a unique number in $\{0, \dots |Q| - 1\}$ where the initial state q_{init} corresponds to 0.

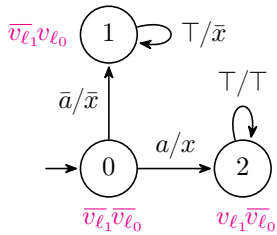
We use a set of $N = \lceil \log_2(|Q|) \rceil$ latches to remember the current state of the machine. For a latch $\ell \in \{\ell_0, \ell_1, \dots, \ell_{N-1}\}$, we write v_ℓ its current Boolean value.

Our first step is to convert the IGMM into a set of Boolean functions that represents the behavior of the controller. That is, each output signal $o \in O$ is associated to a Boolean function f^o that can be computed using the current value of the latches and the current value of each input signal. Similarly, for each ℓ, there exists a function f^ℓ that computes its next value based on the current values of all latches and input signals.

As an example, Figure 15 shows a possible encoding for the IGMM of Figure 5. (We do not use Fig. 6 because the Boolean encoding would be too simple to illustrate the upcoming variants of AIG encoding.)

Algorithm 3 shows how these functions are computed from the input IGMM. Initially, all output and latch functions are initialized to the false formula (lines 3–4). Then for each state q, lines 6–11 compute a Boolean formula f^q representing its encoding using latches. Here, the function BinaryEnc(q) returns the array of $\lceil \log_2(|Q|) \rceil$ Boolean values corresponding to the binary encoding of q. Then the loop on line 12 considers the edges leaving q for each input valuation u: if the output of that edge is not completely specified, an output valuation $v \in \mathbb{B}^O$ is chosen with ChooseOneValuation on line 16. (For instance, in Figure 15 this function chooses to output $\bar{x}$ for any input read from state 2.) For each latch ℓ that should be true in the encoding of the destination state, lines 17–19 add the clause $f^q \wedge u$ to f^ℓ. And similarly, for each output signal o that is true in v, lines 20–22 add that clause to the formula f^o.

In our implementation, these Boolean functions are represented using *Binary Decision Diagrams* (BDD) (Bryant, 1986). Continuing the example of Figure 15, Figure 16 shows how f^{ℓ_0}, f^{ℓ_1} and f^x are stored. These Boolean



$$f^x = \overline{v_{\ell_1}} \wedge \overline{v_{\ell_0}} \wedge a \quad (1)$$

$$f^{\ell_0} = (\overline{v_{\ell_1}} \wedge \overline{v_{\ell_0}} \wedge \bar{a}) \vee (\overline{v_{\ell_1}} \wedge v_{\ell_0}) \quad (2)$$

$$f^{\ell_1} = (\overline{v_{\ell_1}} \wedge \overline{v_{\ell_0}} \wedge a) \vee (v_{\ell_1} \wedge \overline{v_{\ell_0}}) \quad (3)$$

Fig. 15: Possible Boolean encoding of the IGMM of Figure 5. Two latches ℓ_0 and ℓ_1 are used to keep track of the current state of the machine. Function f^x specifies that x should be emitted only if we are in state 0 (encoded with $\overline{v_{\ell_1}} \wedge \overline{v_{\ell_0}}$) and if a is read. It implies that $\bar{x}$ will be output in other cases (therefore this encoding has made the choice to output $\bar{x}$ from state 2). Functions f^{ℓ_0} and f^{ℓ_1} give similar conditions for the values of the latches.

Algorithm 3 ToSymbolicMealy

Input: IGMM $M = (I, O, Q, q_{init})$.
Output: Boolean functions for all outputs and latches.
$\forall j \in \{0, \ldots, N-1\}: \; f^{\ell_j} \leftarrow \bot$
$\forall o \in O: \; f^o \leftarrow \bot$
for $q \in Q$ **do**
    $f^q = \top$
    for $j \in \{0, \ldots, N-1\}$ **do** ▷ Encode q as a conjunction of latches
        if BinaryEnc$(q)[j]$ **then**
            $f^q \leftarrow f^q \wedge v_{\ell_j}$
        else
            $f^q \leftarrow f^q \wedge \bar{v}_{\ell_j}$
    for $u \in \mathbb{B}^I$ **do** ▷ Update all functions for each input
        if $\delta(q, u)$ **is** undefined **then**
            continue
        $q' = \delta(q, u)$
        $v =$ ChooseOneValuation$(\lambda(q, u))$
        for $j \in \{0, \ldots, N-1\}$ **do**
            if BinaryEnc$(q')[j]$ **then**
                $f^{\ell_j} \leftarrow f^{\ell_j} \vee (f^q \wedge u)$
        for $o \in O$ **do**
            if $v(o) = \top$ **then**
                $f^o \leftarrow f^o \vee (f^q \wedge u)$
return $\{f^o \mid o \in O\}, \{f^\ell \mid \ell \in \{\ell_0, \ldots \ell_{N-1}\}\}$

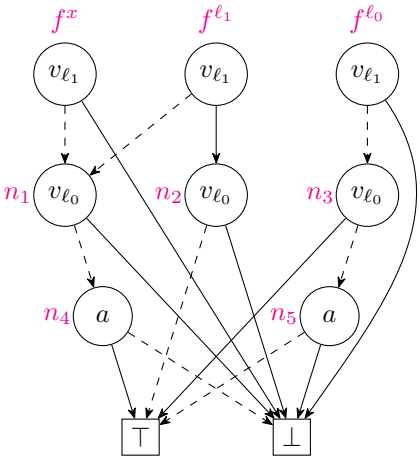


Fig. 16: BDD representation of f^x, f^{ℓ_0}, and f^{ℓ_1} from Figure 15. A BDD is like a decision tree with sharing of identical subtrees. Each round node is labeled by a Boolean variable that is questioned, plain edges should be followed when the variable is true, and dashed edges when it is false. Variables always appear in the same order along a branch.

functions then have to be encoded into an AIG. The next two sections propose two different encodings.

5.2 If-Then-Else Encoding

Our first AIG encoding is inspired from the BDD representation of our functions. Assuming an order over the variables has been fixed, the BDD representation of a function can be seen as an If-The-Else normal form where each node $\overset{v}{\underset{n_1}{\diagup}}\!\!\underset{n_2}{\diagdown}$ can be read as "if v then n_1 else n_2".

This can be naturally encoded as $(v \wedge n_1) \vee (\bar{v} \wedge n_2)$, or, since we should be only using NOT and AND gates, $\overline{\overline{v \wedge n_1} \wedge \overline{\bar{v} \wedge n_2}}$. That expression can of course be simplified when n_1 and n_2 are $\top$ or $\bot$.

Continuing our running example, Figure 17 shows the AIG encoding for each node of Figure 16, and Figure 18 shows the circuit assembled from those parts.

This encoding uses at most three gates per node in the BDD representation of all Boolean functions. It is also sensitive to the ordering of variables in the BDD representation.

5.3 Irredundant-Sum-Of-Products Encoding

An alternative way to encode Boolean functions using AND and NOT gates, is to first rewrite these Boolean functions into disjunctive normal form (also known as sum-of-products). In particular there exist BDD-based algorithms

$$f^x = (v_{\ell_1} \wedge \bot) \vee (\overline{v_{\ell_1}} \wedge n_1) = \overline{v_{\ell_1}} \wedge n_1$$

$$n_1 = (v_{\ell_0} \wedge \bot) \vee (\overline{v_{\ell_0}} \wedge n_4) = \overline{v_{\ell_0}} \wedge n_4$$

$$n_4 = (a \wedge \top) \vee (\bar{a} \wedge \bot) = a$$

$$f^{\ell_0} = (v_{\ell_1} \wedge \bot) \vee (\overline{v_{\ell_1}} \wedge n_3) = \overline{v_{\ell_1}} \wedge n_3$$

$$n_3 = (v_{\ell_0} \wedge \top) \vee (\overline{v_{\ell_0}} \wedge n_5) = \overline{\overline{v_{\ell_0}} \wedge \overline{\overline{v_{\ell_0}} \wedge n_5}}$$

$$n_5 = (a \wedge \bot) \vee (\bar{a} \wedge \top) = \bar{a}$$

$$f^{\ell_1} = (v_{\ell_1} \wedge n_2) \vee (\overline{v_{\ell_1}} \wedge n_1) = \overline{\overline{(v_{\ell_1} \wedge n_2)} \wedge \overline{(\overline{v_{\ell_1}} \wedge n_1)}}$$

$$n_2 = (v_{\ell_0} \wedge \bot) \vee (\overline{v_{\ell_0}} \wedge \top) = \overline{v_{\ell_0}}$$

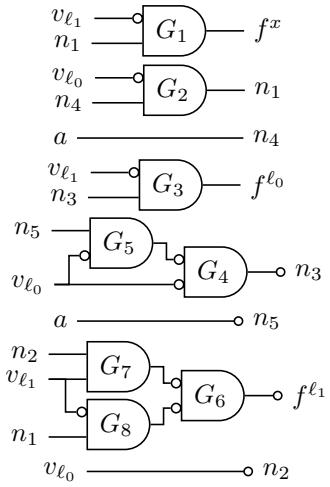


Fig. 17: Step-by-step encoding of functions f^x, f^{ℓ_0}, and f^{ℓ_1} as And-Inverter Graphs, based on the If-Then-Else normal form of these functions, naturally visible in Figure 16.

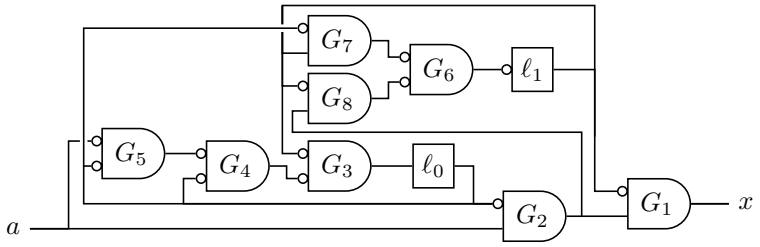


Fig. 18: Full circuit assembled from the bits in Figure 17.

to compute so called *irredundant sum-of-products* (*ISOP*) where each product is a prime implicant that cannot be removed without changing the function (Minato, 1992).

Figures 19–20 show this encoding of our running example.

However, note that (regardless of the encoding used) the encoder can easily detect that a gate computing a combination of signals already exists, and reuse it. This is the case of gate H_2 in Figure 19.

Without the above optimization, this encoding would use as many gates as symbols $\vee$ or $\wedge$ in the ISOP of the encoded function. Each product of the ISOP can be also encoded in multiple ways (Figure 21), and the same is true for the encoding of their sum.

In SYNTCOMP, only the number of gates and latches is taken into consideration to measure the quality of the generated circuit. Another measure that could be used is the delay between the input and output: i.e., the number of gates on the longest path through the circuit. Balancing the binary tree of

$$f^x = \overline{v_{\ell_1}} \wedge \overline{v_{\ell_0}} \wedge a$$

$$f^{\ell_0} = (\overline{v_{\ell_1}} \wedge \bar{a}) \vee (\overline{v_{\ell_1}} \wedge v_{\ell_0}) = \overline{\overline{(\overline{v_{\ell_1}} \wedge \bar{a})} \wedge \overline{(\overline{v_{\ell_1}} \wedge v_{\ell_0})}}$$

$$f^{\ell_1} = (\overline{v_{\ell_0}} \wedge a) \vee (v_{\ell_1} \wedge \overline{v_{\ell_0}}) = \overline{\overline{(\overline{v_{\ell_0}} \wedge a)} \wedge \overline{(v_{\ell_1} \wedge \overline{v_{\ell_0}})}}$$

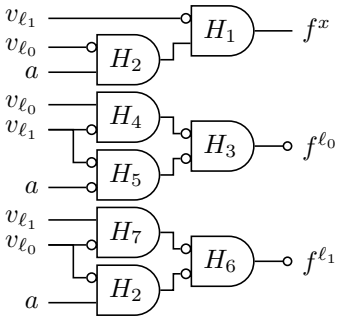


Fig. 19: Encoding of functions f^x, f^{ℓ_0}, f^{ℓ_1} as And-Inverter Graphs based on Irredundant Sums-of-Products.

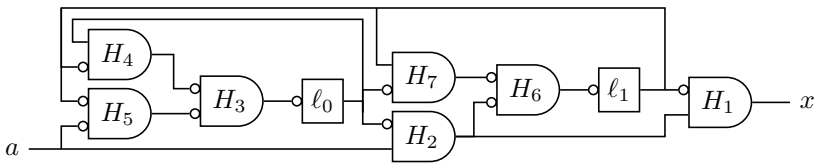


Fig. 20: Full circuit assembled from the bits of Figure 19.

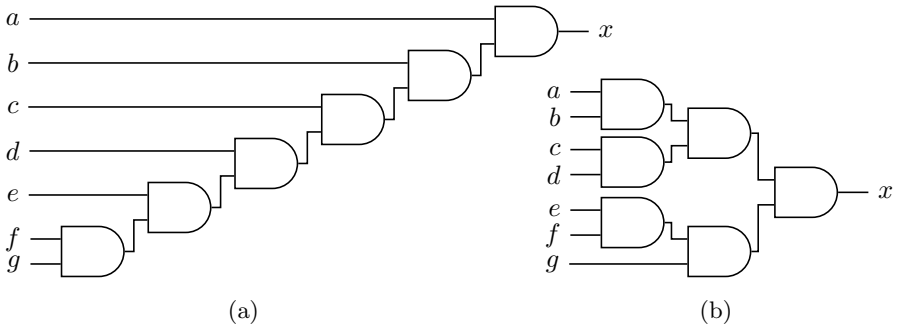


Fig. 21: The product $a \wedge b \wedge c \wedge d \wedge e \wedge f \wedge g$ requires 6 gates that can be arranged as a binary tree of any shape.

gates encoding n-ary conjunctions and disjunctions helps reducing this delay and improving the sharing of gates in the circuit (as will be discussed below).

5.4 Other Improvements to the Encoding

Dual Encoding

An idea to reduce the number of gates needed is to check whether it is easier to encode the function itself or its negation. So instead of encoding $f^x = a \vee b \vee c$

we can encode $f^{\bar{x}} = \bar{a} \wedge \bar{b} \wedge \bar{c}$ and negate the output. This can, depending on the function, lead to substantial gains.

Choosing the Specialization and Using Flexibility

The IGMM used within `ltlsynt` is strictly more expressive than *traditional* Mealy machines, as they associate to a state and input valuation a set of outputs (the elements of which all satisfy the specification). However, the actual circuit cannot represent sets of outputs, but needs to respond to a concrete input valuation with a concrete output valuation. In Algorithm 3, the function ChooseOneValuation picks one of the compatible output valuations, which is then used to construct a circuit.

We use the following heuristic to choose a cube (over the output variables) such that

- All valuations compatible with the cube are compatible with the original condition.
- It minimizes the cost function $\alpha.nhigh + \beta.nlow + \gamma.ndc$ with $nhigh$ being the number of outputs set to $\top$, $nlow$ being the number of outputs set to $\bot$ and ndc being the number of outputs that do not appear in the cube. α, β, γ are fixed non-negative integer values corresponding to costs.

For instance, assume we have $\delta(q, \ell) = xyz \vee x\bar{z}$ for some q and ℓ with $O = \{x, y, z\}$. The compatible cubes are xyz and $x\bar{z}$ with the respective costs 3α and $\alpha + \gamma + \beta$.

We typically use costs such that $\beta < \gamma < \alpha$. The reasoning behind this is simple: we prefer an output that is set to $\bot$ as in such a case there is no need to encode the current transition hopefully leading to fewer gates. The opposite case with the output being set to $\top$ is, usually, the least favorable as here the transition must be encoded. In between we have variables which do not appear (also referred to as *do-not-care* or just *dc* variables) in the cube, which can therefore be set to $\top$ or $\bot$.

Finally, we can use this information about the do-not-care in the following way: for each output o we build two Boolean functions, one denoted f^o which corresponds to the cases where o *must* be set to $\top$. The second one, denoted $f^{o,dc}$, corresponds to cases where o *can* be set to $\top$. We can then proceed to encode f^o and $f^o \vee f^{o,dc}$ and use whichever results in fewer gates.

Increase Gate Sharing

So far we have seen heuristics which try to minimize the number of gates needed to represent a given Boolean function. Another important aspect is to increase the number of gates which are shared between different Boolean functions. Indeed, the symbolic representation of the Mealy machine to be encoded has one function per latch and output. Increasing the number of gates shared between these outputs might therefore yield significant overall reductions.

In the case of the ITE construction, this idea comes down to reordering the variables within the BDD. In order for this to be effective, the BDD variables

would have to be reordered with respect to the Boolean functions to be encoded (and only those). Unfortunately we are currently not able to do this, due to the architecture of Spot.

However, we can use this idea and apply it to the ISOP construction. To this end we split the tree generated for each product (a cube) and arrange it into two subtrees: one connecting latches, and one connecting input variables. The intuition is that such a decoupling should increase the chance of reusing the gate to encode another function: particular valuations of the latches, or valuations of input variables are likely to be reused elsewhere in the circuit.

Other Implementation Details

As shown in Figure 7 (and discussed later in Section 7.3) we seek to decompose the given formula if possible (Finkbeiner et al, 2021). In the case where decomposition is successful, multiple IGMMs have to be encoded jointly into a single AIG.

For the decomposition to be successful, the set of output variables of the different IGMMs built have to form a partition of all controllable propositions. The output of each IGMM can therefore be computed separately without any dependency between the different machines.

We can therefore take the following approach: each Mealy machine is associated to its own set of latches of appropriate size. Then Algorithm 3 is run for each of the machines and the resulting Boolean functions are encoded indifferently.

In this setting, splitting the cubes into sub-cubes over latches and inputs during the ISOP construction can be advantageous as it improves the gate-sharing of expressions over the input variables, which are common to all machines.

In the above we have discussed the different options used to translate a Boolean function into an AIG (ITE, ISOP, dual encoding, splitting propositions). Unfortunately we have not discovered any good heuristic allowing us to choose among these options. We therefore implemented a mechanism which lets us easily test multiple configurations and retain the smallest solution.

When using `ltlsynt`, the encoding is controlled via the `--aiger` option. The argument is a comma-separated list with elements of the form `ite|isop|both[+ud][+dc][+sub0|sub1|sub2]`. The first and only mandatory part determines whether to use the ITE or ISOP construction method. The option `both` will first use ITE, then ISOP and retain the smaller for each Boolean function to encode.

The other options are

`+ud` : enables the use of the dual encoding
`+dc` : enables the use of do-not-care variables
`+sub`X : Controls whether there is a distinction between the variables

$X = 0$: No distinction
$X = 1$: Split cube into sub-cubes over inputs and latches

$X = 2$: Search for common variables[1]

The effects of these options are discussed in Section 7.1.2.

6 Bypassing the Game Theoretical Framework

We are now going to discuss a synthesis method applicable to a subset of LTL formulas. Section 7.2 will show that this subset concerns 15% of the specifications used in SYNTCOMP'19. The method consists of two parts.

The first part is a syntactical analysis of the formula. If the specification is found to belong to a specific subclass of LTL, this analysis can indicate whether the specification is realizable or not, otherwise `ltlsynt` will have to go through the usual construction.

The second part consists in a direct construction of a Mealy machine for the specification, based on some automata-theoretic operations.

The subset of LTL formulas supported by this approach is $\mathsf{G}(b_1) \wedge \underbrace{(\phi_1 \leftrightarrow \phi_2)}_{\psi}$ where b_1 is a *synthetizable* Boolean formula (i.e., there is no input valuation u such that $u \wedge b_1$ is false), and additionally one of the following cases applies:

- ϕ_1 is an LTL formula with only inputs that can be translated into a deterministic Büchi automaton, ϕ_2 is $\mathsf{GF}(b_2)$ where b_2 and $\neg b_2$ are synthetizable Boolean formulas.
- ϕ_1 is an LTL formula with only inputs that can be translated into a deterministic co-Büchi automaton, ϕ_2 is $\mathsf{FG}(b_2)$ where b_2 and $\neg b_2$ are synthetizable Boolean formulas.

As indicated in Theorem 2.5, we know that for any syntactic recurrence formula, there exists a deterministic Büchi automaton. The LTL translation algorithm of Spot, called ToNBA, usually tries to build deterministic automata, but it does not guarantee that any recurrence formula will be translated into a DBA. Algorithm 4 therefore uses the syntactic check as a first filter, and then try its luck with ToNBA to maybe obtain a DBA. The dual is performed on persistence formulas to obtain deterministic co-Büchi automata.

A way to read a formula like $\phi_1 \leftrightarrow \mathsf{GF}(b_2)$ is *If ϕ_1 is verified, then b_2 must be true infinitely often, otherwise b_2 must be verified finitely often.*

Our construction therefore creates an automaton $\mathcal{A}$ for ϕ_1 (line 8) and then adds b_2 to each ⓿-colored edge of this automaton (second case of the f helper function) and $\neg b_2$ to an uncolored edge (third case). At the same time, b_1 is added to all edges to satisfy the $\mathsf{G}(b_1)$ part of the original formula. The colors are completely removed from the final automaton and its acceptance is set to $\top$ to match that of a Mealy machine. (Our implementation then interprets the automaton built by Algorithm 4 as an IGMM.)

Formulas of the shape $\phi_1 \leftrightarrow \mathsf{FG}(\neg b_2)$ are dealt with in a dual manner.

[1] This option was not detailed as it was not found to be better than $X = 1$.

Algorithm 4 Creation of a valid strategy for a subset of LTL formula

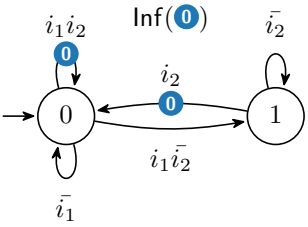
if ϕ does not have the shape $\underbrace{\mathsf{G}(b_1) \wedge (\phi_1 \leftrightarrow \mathsf{GF}(b_2))}_{\text{shape 1}}$ where ϕ_1 is a syntactic recurrence with only inputs, or the shape $\underbrace{\mathsf{G}(b_1) \wedge (\phi_2 \leftrightarrow \mathsf{FG}(\neg b_2))}_{\text{shape 2}}$ where ϕ_2 is a syntactic persistence with only inputs **then**
  return unsupported
if b_1 is not synthetizable **then**
  return unrealizable
if b_2 and $\neg b_2$ are not both synthetizable **then**
  return unsupported
if ϕ has shape 1 **then**
  $\mathcal{A} \leftarrow \mathsf{ToNBA}(\phi_1)$
  if $\mathcal{A}$ is not deterministic **then**
    return unsupported
else if ϕ has shape 2 **then**
  $\mathcal{A}_{\neg\phi_2} \leftarrow \mathsf{ToNBA}(\neg\phi_2)$
  if $\mathcal{A}_{\neg\phi_2}$ is not deterministic **then**
    return unsupported
  $\mathcal{A} \leftarrow \mathsf{Complement}(\mathcal{A}_{\neg\phi_2})$ ▷ A det. co-Büchi automaton for ϕ_2
let $(Q, M, \Sigma, \delta, q_0, \alpha) = \mathcal{A}$
$\delta' \leftarrow \{(s, f(s, \ell, M, d), \varnothing, d) \mid (s, \ell, M, d) \in \delta\}$
where $f(s, \ell, M, d) = \begin{cases} \ell \wedge b_1 & \text{if } \mathsf{SccOf}(s) = \mathsf{SccOf}(d) \\ \ell \wedge b_1 \wedge b_2 & \text{if } \mathsf{SccOf}(s) \neq \mathsf{SccOf}(d) \text{ and } m = \{\textcircled{0}\} \\ \ell \wedge b_1 \wedge \neg b_2 & \text{if } \mathsf{SccOf}(s) \neq \mathsf{SccOf}(d) \text{ and } m = \varnothing \end{cases}$
return $(Q, \varnothing, \Sigma, \delta', q_0, \top)$

Example 1 Consider the formula $(\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o)$. As it is a recurrence formula, $(\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2))$ can be associated with the deterministic Büchi automaton of Fig. 22a. When we associate o with the colored edges and $\bar{o}$ with the others, we then obtain the automaton of Fig. 22b which is a valid strategy for the specification.

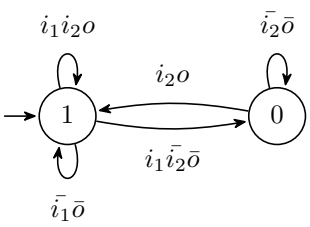
An additional optimization, present in Algorithm 4 is the use of strongly connected components (SCCs). A transition between two different SCCs cannot be seen infinitely often, so it does not need to enforce either b_2 or $\neg b_2$. This "don't care" can potential later improve the IGMM reduction, as discussed in Section 4.

Remark 1 In order to maximize the number of formulas detected by this algorithm, the following rewriting rules can be applied:

- If a formula is of the form $\mathsf{G}(b_1) \wedge \mathsf{G}(b_2) \wedge \ldots \wedge \mathsf{G}(b_n) \wedge \psi$, rewrite it as $\mathsf{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_n) \wedge \psi$,



(a) Büchi automaton associated to $\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)$



(b) Strategy obtained from the automaton of Fig. 22a for $(\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o)$

Fig. 22: Building a direct strategy for $(\mathsf{GF}(i_1) \wedge \mathsf{GF}(i_2)) \leftrightarrow \mathsf{GF}(o)$

- If a formula is of the form $\mathsf{G}(b)$, rewrite it as $\mathsf{G}(b) \wedge (\top \leftrightarrow \mathsf{GF}(\top))$,
- Otherwise, rewrite ψ as $\mathsf{G}(\top) \wedge \psi$.

We can remark that the last rule does not satisfy that b_2 and its negation are both synthetizable. However, $\neg b_2$ has to be realizable in order to avoid removing an edge when working on a non-accepting edge. In this case, the Büchi automaton associated to $\top$ has only one (accepting) edge, so this problem does not occur and the procedure is valid. The same idea can be applied for the second rule.

Appendix A justifies the restrictions on b_1 and b_2, and why ϕ_1 must have only inputs.

7 Some Benchmarks

In this section we evaluate the impact of different options on both the runtime as well as the quality of the result. All results presented can be recreated from the artifact available at https://www.lrde.epita.fr/~frenkin/fmsd22/artifact.

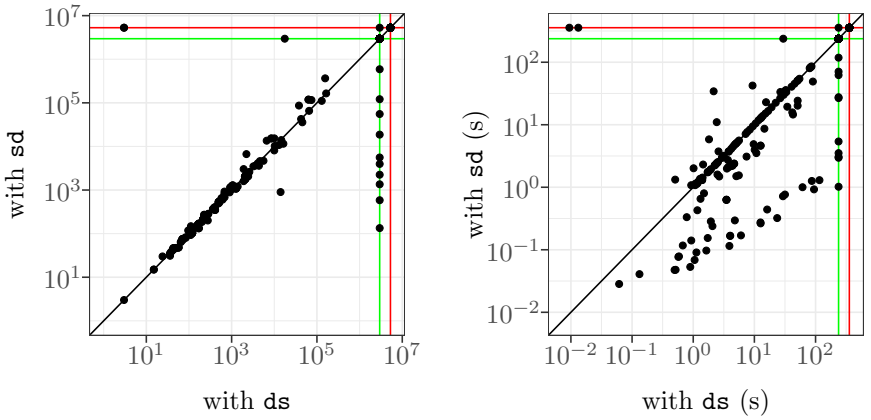
These benchmarks use the development version of `ltlsynt` that was eventually released with version 2.11 of Spot (Duret-Lutz et al, 2022). The LTL specifications we used as input come from the 2019 edition of SYNTCOMP.

7.1 Effect of `--algo` on Parity Game Size

This part focuses on the impact of the choice of the algorithm used to compute the parity game. We compare the four different algorithms `ds`, `sd`, `lar` and `acd` described in Fig. 8. We study the time spent by `ltlsynt` to execute all steps, from the formula to the game.

The measurements are performed twice, then the minimal time is kept with a timeout of 120 seconds.

To avoid a bias induced by numerous small benchmark instances, we only consider formulas for which game construction takes more than one second for at least one of the algorithms.



(a) Comparison of the size of the parity games obtained with the `ds` and `sd` configurations

(b) Minimal total time used by `ltlsynt` with `ds` and `sd` to produce a game

Fig. 23: Comparison of approaches `ds` and `sd` to build a parity game. The dots on the red lines correspond to errors while those on the green lines to timeouts.

The machine used for the measurements features 16GB of RAM and an Intel Core i7-3770 processor.

We first compare `ds` and `sd` in Fig. 23. For both algorithms, there are numerous timeouts, but while it concerns 273 cases for `ds`, we have only 263 such cases for `sd`. Furthermore, while `ds` leads to an error[2] for 19 cases, it happens for 22 cases with `sd`. However, for one case that raises an error with `sd`, there was a timeout for `ds`.

A comparison of the number of states shows us that `ds` produces a larger game than `sd` in 51 cases. Conversely, for 43 cases `sd` produces a larger game. Therefore, the results of the two algorithms are comparable.

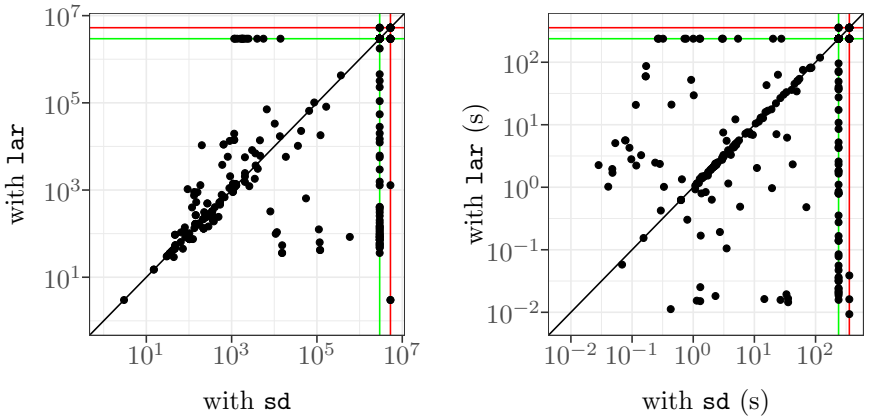
Let us now compare processing speed. We say that an algorithm X is faster than an algorithm Y if Y takes at least 10% longer to complete the task. Moreover, we only consider cases for which both algorithms terminate successfully.

Based on these definitions, `ds` is faster for 12 cases while `sd` is faster for 67 cases. Fig. 23b is clearly favorable to `sd`. This matches the intuition given in Section 3.1 that the determinization is more efficient if split is performed beforehand.

We can conclude that parity games produced by both approaches have similar size on average, but there is a runtime advantage in favor of `sd`.

Now, let us compare `sd` with `lar` in Fig. 24. The first thing we notice is that we have 263 timeouts and 22 errors with `sd`, whereas with `lar` 238 timeouts and 13 errors occur. In particular, we have 9 cases where there is a timeout

[2] An error in this context is either insufficient memory or the acceptance condition needing more than 32 colors, the default number of colors in Spot.



(a) Comparison of the size of the parity games obtained with the `sd` and `lar` configurations

(b) Minimal total time used by `ltlsynt` with `sd` and `lar` to produce a game

Fig. 24: Comparison of approaches `sd` and `lar` to build a parity game.

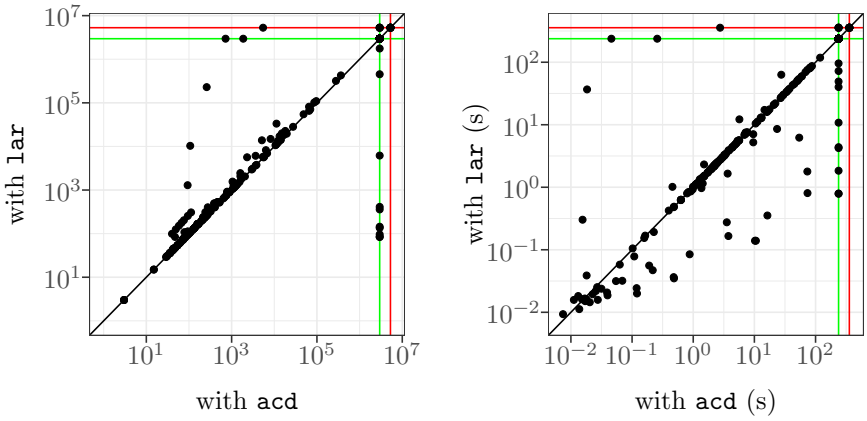
with `lar` but an error with `sd` and conversely 3 cases where `sd` has a timeout while `lar` gives an error.

Let us study the number of states. We see that while there are 55 cases where a smaller machine is produced with `lar`, there are 53 cases where `sd` performs better.

Comparing their respective runtimes is hard. Even if there exists a set of cases for which the two algorithms give close enough results, there are also some sets of cases for which each of the algorithms performs really faster.

In conclusion, these two algorithms are difficult to compare, both in terms of size of the resulting machine and of run time. However, `lar` is able to solve 34 more cases than `sd`.

The last comparison concerns `lar` and `acd`; it is shown in Fig. 25. While we have 238 timeouts and 13 errors with `lar`, we have 250 timeouts and 8 errors with `acd`. We have 4 cases where `acd` exceeds the maximum duration but `lar` makes an error while the opposite never happens. If we look at the number of states, there are 83 cases where `acd` produces a smaller machine while there are 5 cases where `lar` gives a better result. `acd` and `lar` only differ in the paritization procedure and the ACD-transform cannot give a bigger parity automaton than the paritization procedure of `lar`. The set of cases favorable to `lar` comes from the splitting performed on these parity automata which depends on the coloring of the automata (this suggests that our splitting procedure could probably be improved to better support the type of coloration produced by the ACD-transform, or conversely that the ACD-transform could be fine-tuned to this case).



(a) Comparison of the size of the parity games obtained with the `acd` and `lar` configurations

(b) Minimal total time used by `ltlsynt` with `acd` and `lar` to produce a game

Fig. 25: Comparing between `acd` and `lar` the run time of the computation of a parity game. The dots on the red lines correspond to errors while those on the green lines correspond to timeouts.

While `lar` was faster in 35 cases, `acd` was faster in 11 cases. However, in general, when `acd` is faster, it is close to `lar`, while we can find many cases where `lar` is significantly faster than `acd`.

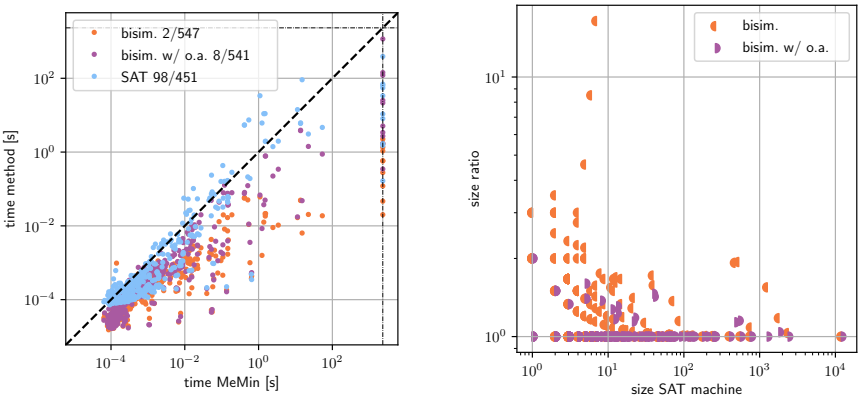
We can conclude that these algorithms give close enough results but `lar` is faster, even if it produces bigger machines.

Even though each algorithm has its advantages, we will keep `lar` as the default algorithm.

7.1.1 Comparison of Reduction Methods

We merely remind the reader of the conclusions of a comparison previously published (Renkin et al, 2022), restricting the cases studied to those of SYNTCOMP'19. We benchmark our methods against MeMin, a state-of-the-art minimization tool for incompletely specified Mealy machines relying on a reduction to SAT (Abel and Reineke, 2015). Fig. 26a shows that our methods are faster than MeMin; in particular, *bisimulation with output assignment* is faster than our SAT-based method. Furthermore, we can see thanks to Fig. 26 that *bisimulation with output assignment* was able to give a solution close to the optimum for many cases of the SYNTCOMP sample. This is the reason why this reduction is the default option in `ltlsynt`.

Fig. 26 displays a comparison of our three reductions with MeMin. We clearly see in Fig. 26a that our reductions are faster than this tool. Fig. 26b proves that even if they are mere heuristics, *bisimulation* and *bisimulation with output assignment* both remain fairly close to an optimal solution.



(a) Total runtime for instances derived from SYNTCOMP.

(b) Size ratios for instances derived from SYNTCOMP.

Fig. 26: Details for SYNTCOMP'19 instances

7.1.2 Comparison of AIG-Encodings

As presented in Section 5, `ltlsynt` provides several options in order to determine how the AIG is derived from a strategy. We will focus here on the time it takes to generate the AIG as well as the resulting number of gates in said circuit. The number of latches always being the same, these two metrics are the only ones of interest with respect to SYNTCOMP.

To perform these benchmarks, we pre-generated 780 non-trivial strategies[3] from the set of SYNTCOMP'19 benchmarks by using `ltlsynt`. To obtain said strategies, we used approaches `lar` and `ds`. We did not seek to decompose the formulas, nor did we try to bypass the game-theoretical framework.

Results are presented in Table 2. Remember that $+ud$ stands for the dual encoding option (encode the negation of a function as well as the function itself and retain the smaller solution), $+dc$ implies the use of do-not-cares (i.e. tries to leverage the flexibility in some outputs), and $+sub1$ indicates that we treat variables which correspond to latches separately from those corresponding to input variables.

- ITE construction is on average 3 to 4 times faster than the ISOP.
- ISOP construction usually generates smaller circuits than ITE.
- In some cases, ITE produces significantly smaller circuits than ISOP.
- For ITE, the options $+dc$ and $+ud$ have a minor positive effect on circuit size while having a minor negative impact on encoding time.
- For ISOP, the positive effect of $+dc$ and $+ud$ is significantly higher, but so is the negative effect on encoding time.

[3]Strategies which cannot be encoded without gates.

	tot. time	mean	gmean	median
ite	248	28529	102.4	66.0
ite + *ud*	281	28527	97.1	65.5
isop + *sub*0	821	8535	81.0	60.0
isop + *dc* + *ud* + *sub*1	1056	7510	66.8	47.0
*ite*2*isop*	1247	6159	65.7	47.0

Table 2: Results for different encoding options presenting the cumulative time taken to encode all strategies (tot. time, in seconds) as well as the mean, geometric mean, and median of the number gates in the resulting AIG. *ite* and *isop* give the baseline for each construction. *ite*+*dc*+*ud* and *isop*+*dc*+*ud*+*sub*1 are the options resulting in the smallest circuits for each construction method. *ite*2*isop* corresponds to *ite* + *ud*, *isop* + *dc* + *ud* + *sub*1. It uses the better circuit that can be obtained from the two.

	<(1)	<(2)	<(3)	<(4)	<(5)
(1) *ite*		33	210	75	3
(2) *ite* + *ud*	232		249	90	0
(3) *isop* + *sub*0	506	429		74	45
(4) *isop* + *dc* + *ud* + *sub*1	648	590	587		0
(5) *ite*2*isop*	651	590	620	90	

Table 3: The integer in the ith row and jth column corresponds to the number of instances for which the i-th configuration features strictly less gates than the j-th configuration. As one can see, the combination of method (1) and (3) performs the best overall, but due to the intricate relation between the options and the resulting circuit, in 45 cases the base configuration *isop* + *sub*0 still performs better.

Table 3 compares the performances of various encodings. We can see that trying ITE then ISOP is often a good strategy.

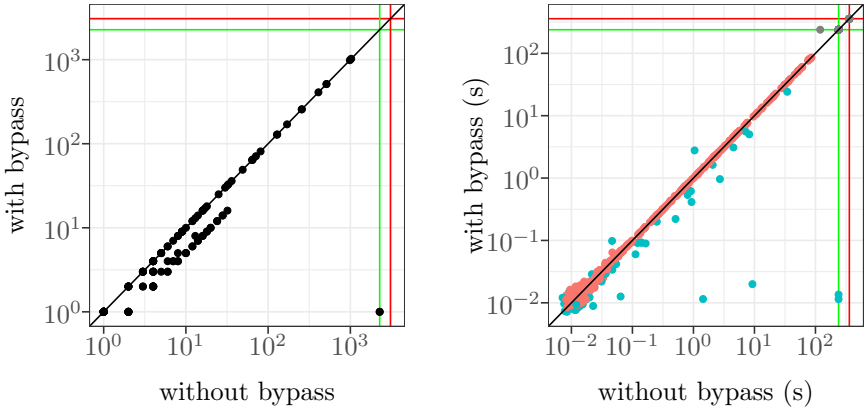
7.2 Interest of Bypassing the Game Construction

The measurements are again performed twice: the minimal result is kept, and the maximal time allowed is 120s. A bisimulation with output assignment is performed for both methods.

First, note that this procedure can be applied to 144 cases out of the 945 SYNTCOMP cases. Focusing on this subset, we show a comparison of the number of states of the strategies obtained with `lar` and with the bypass in Figure 27a.

We can notice that for 103 cases, the strategy yielded by the bypass has at most 10 states, while there are only 10 cases where the strategy has more than 100 states (with a maximum of 1024).

As the bypass never results in a bigger strategy than `lar`, we now focus on the set of 51 cases where the bypass outperforms `lar`. This set of strategies only contains machines with at most 16 states for the bypass method.



(a) Number of states with and without bypass on the set of cases that can be bypassed

(b) Minimal total time spent by `ltlsynt` with algorithm `lar` and the bypass to compute a strategy. Blue dots are cases that can be bypassed, and gray dots, cases where the bypass computation was not able to finish

Fig. 27: Comparing the computation of the strategy between `lar` and the bypass. The dots on the red lines correspond to errors, and those on the green lines, to timeouts.

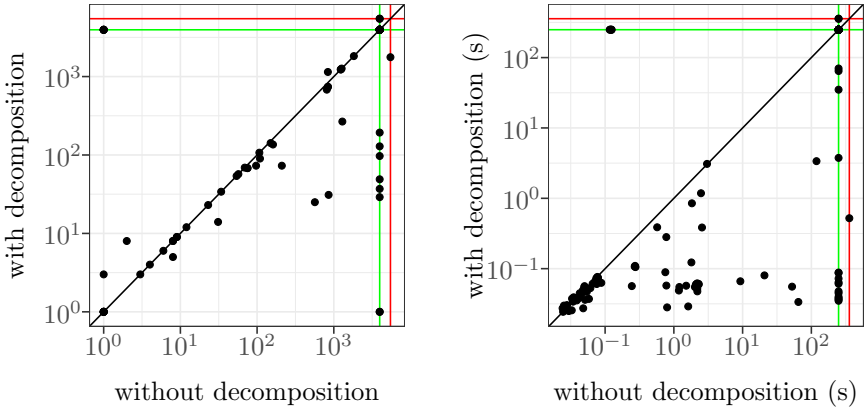
A relevant set of cases is composed of files named `detectorX` where `X` is an integer and the matching formula is of the form $\mathsf{GF}(o) \leftrightarrow \bigwedge_{i\in[1,X]} \mathsf{GF}(i_k)$. On these formulas, the bypass yields a strategy with X states, where each state describes which input variables have been seen. On the other hand, `lar` always returns a strategy that is twice as big.

Even if it is not our default configuration, we can also compare `acd` and the bypass. There is not a single case where the bypass results in a worst strategy whereas `acd` gives a bigger result in 47 cases.

We can also consider Figure 27b that compares the impact on runtime of testing whether the bypass can be used or not. We can see that it seldom increases the runtime even if we don't apply the construction. It is not surprising as such a decision relies on a simple syntactical analysis.

Let us now focus on the set of cases where such a construction is possible. If we exclude the set of cases that are trivially fast with or without the bypass, there remain a fair amount of cases just below the diagonal. Furthermore, two cases can be constructed with the bypass but not with `lar`.

We can conclude that this construction always yields a result that may be better but is never worse than `acd` or `lar` while barely increasing the runtime.



(a) Number of AND gates + 1 with and without decomposition

(b) Minimal time spent by `ltlsynt` to get an AIG circuit with and without decomposition

Fig. 28: Comparing the computation of the AIG circuit with and without the decomposition on the set of formulas that can be decomposed. The dots on the red lines correspond to errors, and those on the green lines, to timeouts.

7.3 Impact of Decomposition

We now focus on the set of 122 formulas that can be decomposed. Even though there are between 2 and 64 sub-specifications, there only remain between 2 and 10 sub-specifications in 89% of the cases.

As we can see in Fig. 28, we found 16 cases that `ltlsynt` could only solve using decomposition. However, the use of this option also leads to the appearance of 5 cases which can no longer be solved.

If we consider only the set of cases for which the two configurations have ended, we see that there are only 3 cases where we obtain bigger circuits with decomposition than without. Conversely, the decomposition yields 15 smaller circuits.

This option results in considerable processing speed improvements. Even if 6 cases end up being slower, these are all solved in less than 0.06s anyway. On the other hand, the computation is significantly faster in 43 cases. The most egregious case is `narylatch10`: the activation of the decomposition reduces the processing time from 64.72s to 0.03s.

We can conclude that, while decomposing a formula may in rare cases prevent us from solving a case in its allotted time, this optimization nonetheless yields, when it is applicable, significant processing speed and size improvements.

8 Conclusion

We have presented the architecture of `ltlsynt`, a tool for synthesizing AIG circuits from LTL formulas. While our approach follows a straightforward automata-theoretic construction, we have described several unique improvements to it.

In particular, we have shown that we can bypass the standard construction for some specific classes of formulas. We have discussed and evaluated various ways to encode Mealy machines as AIG circuits. We have detailed how Spot translates LTL formulas to DELA, largely generalizing the approach of the `delag` tool.

Finally, we have provided numerous benchmarks to compare our various options and approaches as well as justify our default choices.

References

Abel A, Reineke J (2015) MeMin: SAT-based exact minimization of incompletely specified Mealy machines. In: Proceedings for the 34th International Conference on Computer-Aided Design (ICCAD'15). IEEE Press, pp 94–101, https://doi.org/10.1109/ICCAD.2015.7372555

Babiak T, Badie T, Duret-Lutz A, et al (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, vol 7976. Springer, pp 81–98, https://doi.org/10.1007/978-3-642-39176-7_6

Babiak T, Blahoudek F, Duret-Lutz A, et al (2015) The Hanoi Omega-Automata Format. In: Proceedings of the 27th Conference on Computer Aided Verification (CAV'15), Lecture Notes in Computer Science, vol 8172. Springer, pp 442–445, https://doi.org/10.1007/978-3-319-21690-4_31, see also http://adl.github.io/hoaf/.

Biere A (2007) The aiger and-inverter graph (aig) format version 20070427

Brayton R, Mishchenko A (2010) Abc: An academic industrial-strength verification tool. In: Proceedings of the 22nd Conference on Computer Aided Verification (CAV'10). Springer, pp 24–40, https://doi.org/10.1007/978-3-642-14295-6_5

Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8):677–691

Casares A, Colcombet T, Fijalkow N (2021) Optimal transformations of games and automata using Muller conditions. In: Bansal N, Merelli E, Worrell J (eds) Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21), Leibniz International Proceedings

in Informatics (LIPIcs), vol 198. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp 123:1–123:14, https://doi.org/10.4230/LIPIcs.ICALP.2021.123

Casares A, Duret-Lutz A, Meyer KJ, et al (2022) Practical applications of the Alternating Cycle Decomposition. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, to appear

Černá I, Pelánek R (2003) Relating hierarchy of temporal properties to model checking. In: Rovan B, Vojtáš P (eds) Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'03), Lecture Notes in Computer Science, vol 2747. Springer-Verlag, Bratislava, Slovak Republic, pp 318–327

Dax C, Eisinger J, Klaedtke F (2007) Mechanizing the powerset construction for restricted classes of ω-automata. In: Namjoshi KS, Yoneda T, Higashino T, et al (eds) Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07), Lecture Notes in Computer Science, vol 4762. Springer, https://doi.org/10.1007/978-3-540-75596-8_17

van Dijk T (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, https://doi.org/10.1007/978-3-319-89960-2_16

Duret-Lutz A (2014) LTL translation improvements in Spot 1.0. International Journal on Critical Computer-Based Systems 5(1/2):31–54. https://doi.org/10.1504/IJCCBS.2014.059594

Duret-Lutz A, Renault E, Colange M, et al (2022) From Spot 2.0 to Spot 2.10: What's new? In: Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22), Lecture Notes in Computer Science, vol 13372. Springer, pp 174–187, https://doi.org/10.1007/978-3-031-13188-2_9

Emerson EA, Lei CL (1987) Modalities for model checking: Branching time logic strikes back. Science of Computer Programming 8(3):275–306. https://doi.org/10.1016/0167-6423(87)90036-0

Esparza J, Křetínský J, Sickert S (2018) One theorem to rule them all: A unified translation of LTL into ω-automata. In: Dawar A, Grädel E (eds) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18). ACM, pp 384–393, https://doi.org/10.1145/3209108.3209161

Etessami K, Holzmann GJ (2000) Optimizing Büchi automata. In: Palamidessi C (ed) Proceedings of the 11th International Conference on Concurrency Theory (Concur'00), Lecture Notes in Computer Science, vol 1877. Springer-Verlag, Pennsylvania, USA, pp 153–167

Finkbeiner B, Geier G, Passing N (2021) Specification decomposition for reactive synthesis. In: Proceedings for the 13th NASA Formal Methods Symposium (NFM'21), to appear. https://arxiv.org/abs/2103.08459

Jacobs S, Bloem R, Brenguier R, et al (2017) The first reactive synthesis competition (syntcomp 2014). International journal on software tools for technology transfer 19(3):367–390

Jacobs S, Bloem R, Colange M, et al (2019) The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR abs/1904.07736

Jurdziński M (2000) Small progress measures for solving parity games. In: Proceedings of the 17th Symposium on Theoretical Aspects of Computer Science (STACS 2000), Lecture Notes in Computer Science, vol 1770. Springer-Verlag, pp 290–301

Kupferman O, Rosenberg A (2010) The blowup in translating LTL to deterministic automata. In: Revised Selected and Invited Papers for the 6th International Workshop, on Model Checking and Artificial Intelligence (MoChArt'10), Lecture Notes in Computer Science, vol 6572. Springer, pp 85–94, https://doi.org/10.1007/978-3-642-20674-0_6, URL https://doi.org/10.1007/978-3-642-20674-0_6

Löding C (2001) Efficient minimization of deterministic weak ω-automata. Information Processing Letters 79(3):105–109. https://doi.org/10.1016/S0020-0190(00)00183-6

Major J, Blahoudek F, Strejcek J, et al (2019) `ltl3tela`: LTL to small deterministic or nondeterministic Emerson-Lei automata. In: Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis (ATVA'19), Lecture Notes in Computer Science, vol 11781. Springer, pp 357–365, https://doi.org/10.1007/978-3-030-31784-3_21

Manna Z, Pnueli A (1990) A hierarchy of temporal properties. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC'90). ACM, New York, NY, USA, pp 377–410

Minato S (1992) Fast generation of irredundant sum-of-products forms from binary decision diagrams. In: Proceedings of the third Synthesis and Simulation and Meeting International Interchange workshop (SASIMI'92), Kobe, Japan, pp 64–73

Mishchenko A, Chatterjee S, Brayton R (2006) Dag-aware aig rewriting a fresh look at combinational logic synthesis. In: Proceedings of the 43rd Annual Design Automation Conference. Association for Computing Machinery, New York, NY, USA, DAC '06, p 532–535, https://doi.org/10.1145/1146909.1147048, URL https://doi.org/10.1145/1146909.1147048

Müller D, Sickert S (2017) LTL to deterministic Emerson-Lei automata. In: Bouyer P, Orlandini A, Pietro PS (eds) Proceedings of the Eighth International Symposium on Games, Automata, Logics and Formal Verification (GandALF'17), pp 180–194, https://doi.org/10.4204/EPTCS.256.13

Pfleeger CP (1973) State reduction in incompletely specified finite-state machines. IEEE Transactions on Computers C-22(12):1099–1102. https://doi.org/10.1016/j.compeleceng.2006.06.001

Redziejowski R (2012) An improved construction of deterministic omega-automaton using derivatives. Fundamenta Informaticae 119(3-4):393–406. https://doi.org/10.3233/FI-2012-744

Renkin F, Duret-Lutz A, Pommellet A (2020) Practical "paritizing" of Emerson-Lei automata. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20), Lecture Notes in Computer Science, vol 12302. Springer, pp 127–143, https://doi.org/10.1007/978-3-030-59152-6_7

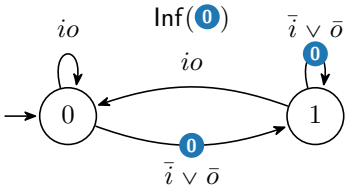
Renkin F, Schlehuber-Caissier P, Duret-Lutz A, et al (2022) Effective reductions of Mealy machines. In: Proceedings of the 42nd International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'22). Springer, Lecture Notes in Computer Science, to appear

Safra S, Vardi MY (1989) On ω-automata and temporal logic. In: Proceedings of the twenty-first annual ACM Symposium on Theory of Computing (STOC'89). ACM, pp 127–137, https://doi.org/10.1145/73007.73019

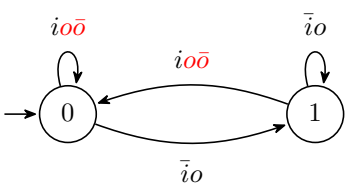
Zielonka W (1998) Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200(1):135–183. https://doi.org/10.1016/S0304-3975(98)00009-7

A Explanation of Restrictions of the Direct Construction

In this appendix, we explain why some restrictions have to be applied to the sub-formulas. Note that these restrictions sometimes allow us to create a direct strategy, but as our procedure is based on syntactical analysis, we may not be able to find all such cases.

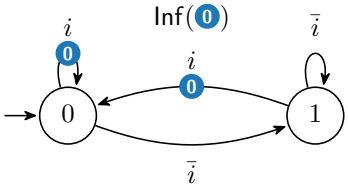


(a) Büchi automaton associated with $\mathsf{GF}((i \wedge \bar{o}) \vee \bar{i})$

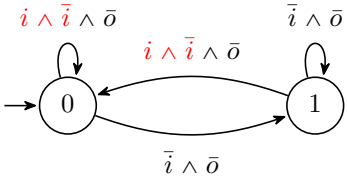


(b) Strategy yielded by the automaton of Figure 29a for the formula $\mathsf{G}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(o)$

Fig. 29: Building a wrong direct strategy for the formula $\mathsf{G}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(o)$



(a) Büchi automaton associated with $\mathsf{GF}(i)$



(b) Strategy yielded by the automaton of Figure 30a for the formula $(\mathsf{GF}(i)) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$

Fig. 30: Building a wrong direct strategy for the formula $(\mathsf{GF}(i) \wedge \mathsf{GF}(\bar{i})) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$

A.1 Why We Don't Have an Output in ϕ_1

Consider the formula $\mathsf{G}((i \wedge \bar{o}) \vee \bar{i}) \leftrightarrow \mathsf{GF}(\bar{o})$. A possible corresponding Büchi automaton is shown in Figure 29. Our procedure would assign $\bar{o}$ to the edge that goes from 1 to 0 and result in an automaton where the states have no outgoing edge labeled by i. Thus, it is not an actual strategy.

A.2 Why b_2 and $\neg b_2$ Must Be Realizable

We now explain why b_2 and its negation have to be realizable. Let us consider an example where b_2 is unrealizable (without loss of generality thanks to symmetry): the formula $\mathsf{GF}(i) \leftrightarrow \mathsf{GF}(\bar{i} \wedge o)$ where o is an output proposition.

The translation of $\mathsf{GF}(i)$ results in a Büchi automaton with two states shown in Figure 30a. The procedure will assign $\bar{i} \wedge o$ to the edges colored with 0 and the condition associated to these edges is now $\bot$; these edges are therefore removed. We end up with a "strategy" described in Figure 30b featuring two states labeled with $\bar{i} \wedge \bar{o}$; it is not input-complete, thus it is not an actual strategy.