Fast Obligation Translation and Synthesis

Alexandre Duret-Lutz[1] (✉), Giuseppe De Giacomo[2], Marcin Jurdzinski[3], Nir Piterman[4], Moshe Y. Vardi[5], and Shufang Zhu[6]

[1] LRE, EPITA, Le Kremlin-Bicêtre, France
[2] University of Oxford, Oxford, UK
[3] DIMAP, University of Warwick, Coventry, UK
[4] University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden
[5] Rice University, Houston, Texas, USA
[6] University of Liverpool, Liverpool, UK

Abstract. Syntactic obligations are a fragment of LTL formulas that translate to deterministic weak ω-automata (DWA). We show that syntactic obligations can be very efficiently converted to minimal DWA represented using multi-terminal binary decision diagrams (MTBDDs), and that synthesis of such specifications can be solved directly on the MTBDD representation on the fly. Our implementation in Spot shows substantial runtime improvements in translation and synthesis.

1 Introduction

Deterministic ω-automata are a key ingredient to many formal methods [69]. They also play a key role in *reactive synthesis*, which takes temporal specifications, typically in Linear Temporal Logic (LTL) [59], and converts them to strategies that ensure their satisfaction [60]. However, the computational complexity and the complex algorithms that are involved in synthesis and, as a consequence, the capacity of tools that support synthesis techniques in their full generality, have proven to be a barrier for implementation [42,29]. Two major successes in this field have been the syntactic fragment of LTL called GR[1] [58], and LTL_f [17], which is a semantic variant of LTL that considers finite rather than infinite computations. In both cases, it is possible to use efficient techniques for handling sets of states, leading to increased capacity. Furthermore, algorithmic techniques for analysis of state spaces and translation of specifications to deterministic automata are simpler [7,18]. In particular, in the context of LTL_f synthesis, tools leverage the algorithmic simplicity of deterministic finite automata (DFA), in contrast to ω-automata [70,4,16,44].

Obligation properties [50], which can be expressed as a boolean combination of *safety* and *guarantee* properties, are good candidates for a fragment of LTL where we could hope to achieve efficient translation and synthesis, exploiting the simplicity of deterministic automata. Although limited in their expressivity, obligation properties are very important in practice. Specifications that are used in model checking or synthesis are often simple (safety, guarantee, and obligation). For instance, the 55 patterns of Dwyer et al. [25] contain 40 obligations. Similarly, Somenzi & Bloem's compilation of 25 LTL formulas "found in the literature" [67, Table 1] contains 16 obligations. Out of the 959 unique LTL specifications collected by the Synthesis Competition [37] at the

Cf. App. A

time of writing, at least 262 specifications are obligations, so more than 27%. Efficient handling of obligation properties also benefits all cases where an obligation property appears as part of a Boolean combination with other temporal properties [65]. Obligation properties are theoretically important, forming the second level in the safety-progress hierarchy of temporal properties [50,51,52,53]. Recently, the safety-progress hierarchy has attracted renewed interest thanks to a normalization procedure that reduces the nesting depth of strong and weak operators [27].

Safety and guarantee properties can be mapped to very simple *deterministic* ω-automata [10]: a deterministic ω-automaton for a safety property only rejects runs that can reach a rejecting sink, and a deterministic ω-automaton for a guarantee only accepts runs that can reach an accepting sink. Boolean combinations of safety and guarantee can be mapped to *deterministic weak automata* (DWA) [45,46], in which each *strongly connected component* (SCC) contains only rejecting cycles or only accepting cycles. The simple nature of DWAs makes them very easy to use: they are closed under Boolean operations using algorithms similar to those of DFAs, and, after a preprocessing stage with linear-cost [46], they can also be minimized like DFAs.

Cf. App. B

Unfortunately, testing whether an LTL formula is an obligation property is quite nontrivial in general [15]. There are, however, well-known syntactic fragments of LTL that capture safety and guarantee properties. By considering their Boolean combinations, we get a syntactic fragment of LTL which we shall refer to as *syntactic obligation* [11,10]. Importantly, every LTL formula representing an obligation property has an equivalent syntactic-obligation formula [11]. Furthermore, many of the previously mentioned examples are syntactic-obligation formulas.

In this work, we use, on the one hand, the simple syntactic structure of syntactic obligations to easily identify obligation properties and, on the other hand, the DFA-like nature of DWAs to improve the translations and synthesis procedures for them. We generalize our recent work on a compact automaton representation of DFAs using Multi-Terminal Binary Decision Diagrams (MTBDDs), that enabled more efficient $\mathrm{LTL_f}$ translation and synthesis [24]. We show that syntactic obligations can be translated efficiently into DWAs represented using MTBDDs, and that the synthesis problem for syntactic obligations can be solved by interpreting the structure of the MTBDDs as a *weak Büchi game* that can be constructed on-the-fly.

We implemented this approach in Spot, a C++ library for LTL and ω-automata algorithms [23]. Our empirical evaluation shows that it significantly improves the generation of deterministic automata for syntactic-obligation properties as well as the synthesis performance for this class of properties. It is worth noting that by implementing efficient obligation translation and synthesis in Spot, we improve the performance of *all* tools building upon Spot whenever they deal with syntactic obligations.

2 Succinct Representation of Deterministic Büchi Automata

In this section, we introduce a representation of deterministic Büchi automata (DBA) based on multi-terminal binary decision diagrams (MTBDDs).

The left-hand side of Figure 1 shows a deterministic Büchi automaton over the set of atomic propositions $\mathcal{P} = \{i_1, i_2, o\}$. The *letters* read by the automaton are assignments

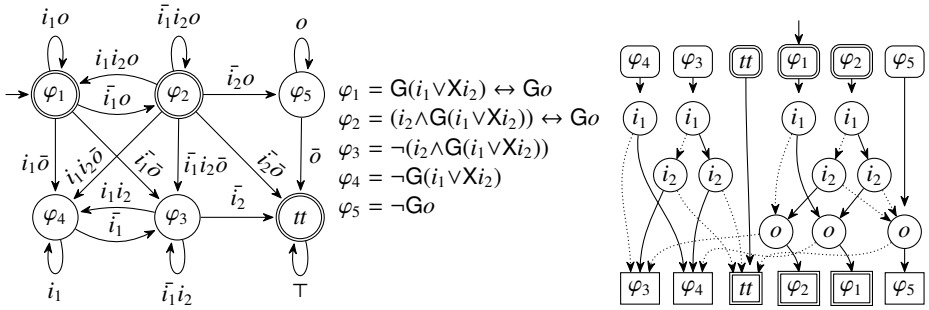

Fig. 1: A DBA for the formula $\varphi_1 = \mathsf{G}(i_1 \vee \mathsf{X}i_2) \leftrightarrow \mathsf{G}o$, and its MTBDD representation.

of all propositions, so there are $2^{|\mathcal{P}|}$ of them, but to simplify the notations, we label the edges of the automaton by Boolean formulas (in this example, implicit conjunctions) that are satisfied by all the assignments that would normally label the edge. In Spot, this explicit representation of automata uses edges labeled by BDDs. In this example, we have also named the states of the automaton using LTL formulas, but this is merely decorative. This DBA has four strongly-connected components: $\{\varphi_1, \varphi_2\}$, $\{\varphi_3, \varphi_4\}$, $\{\varphi_5\}$, and $\{tt\}$. The automaton is *weak* because each SCC contains only accepting states, or only rejecting states. We abbreviate *weak DBA* as *DWA*.

The right-hand side of Figure 1 shows a semi-symbolic representation of the same DBA, where the outgoing transitions of each state are represented using MTBDDs. The representation is inspired from the representation of DFAs in Mona [35,38], and previous work [24]. The reader familiar with binary decision diagrams (BDDs) [8] will recognize the classical structure of a forest of BDDs: $(x) \overset{\ell}{\underset{h}{\rightarrow}}$ should be interpreted as "if x then h else ℓ", atomic propositions appear in the same order on all paths, and identical subtrees are shared. Contrary to BDDs where leaves (or terminals) are restricted to the values true or false, MTBDDs [47,54,31,38] support arbitrary values: in our case, we encode the name of destination states in the leaves. Each root represents a function from Boolean assignments of atomic propositions to destination states, corresponding to transitions of the automaton. A set of "root pointers" represented by nodes of the form $(s)\rightarrow$ point to an MTBDD representing the outgoing transitions of state s.

Writing $\mathsf{MTBDD}(\mathcal{P}, \mathcal{S})$ to denote an MTBDD with variables in $\mathcal{P}$ and terminals of type $\mathcal{S}$, let us define the MTBDD-based representation of a DBA (MTDBA for short).

Definition 1 (MTDBA, MTDWA). *An MTDBA is a tuple $\mathcal{A} = \langle Q, \mathcal{P}, \iota, \Delta, \lambda \rangle$, where Q is a finite set of* states*, $\mathcal{P}$ is a finite (and ordered) set of* variables*, $\iota \in Q$ is the initial state, $\Delta : Q \to \mathsf{MTBDD}(\mathcal{P}, Q)$ represents the outgoing edges of each state, and $\lambda : Q \to \mathbb{B}$ represents the acceptance status of each state. If the MTDBA is weak, we may call it MTDWA.*

Each path from a root pointer to a terminal in the MTDBA representation on the right-hand side of Figure 1 corresponds to an edge in the explicit DBA represented on the left-hand side. The MTDBA representation is more compact because it can share

common subtrees. For instance, if two states have exactly the same outgoing transitions, but differ only by their acceptance, their root pointers will designate the same MTBDD.

MTBDDs support unary and binary operations using algorithms similar to those of BDDs. For instance, given two $x, y \in \mathsf{MTBDD}(\mathcal{P}, \mathcal{S})$, and a binary operation $\odot : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ over the terminals, one can compute $z \in \mathsf{MTBDD}(\mathcal{P}, \mathcal{S})$, such that for all assignments v of all variables in $\mathcal{P}$, the terminal $z(v)$ reached in z by following v is equal to $x(v) \odot y(v)$. We can therefore lift the $\odot$ operation to the MTBDDs and write $z = x \odot y$.

3 From Syntactic Obligations to MTDWA

This section shows how to build an MTDWA for a syntactic obligation. Our construction is relatively straightforward, yet we could not find any prior work for that fragment.

3.1 Syntactic Sub-Classes of LTL

We assume that the reader is familiar with LTL [59]. The following grammar (where parentheses have been omitted, and $p \in \mathcal{P}$) defines four syntactic fragments [11,10]:

$$\begin{aligned}
\varphi_B &::= \mathit{ff} \mid \mathit{tt} \mid p \mid \neg\varphi_B \mid \varphi_B \wedge \varphi_B \mid \varphi_B \vee \varphi_B \mid \varphi_B \leftrightarrow \varphi_B \mid \varphi_B \oplus \varphi_B \mid \varphi_B \to \varphi_B \mid \mathsf{X}\varphi_B \\
\varphi_G &::= \varphi_B \mid \neg\varphi_S \mid \varphi_G \wedge \varphi_G \mid \varphi_G \vee \varphi_G \mid \varphi_S \to \varphi_G \mid \mathsf{X}\varphi_G \mid \mathsf{F}\varphi_G \mid \varphi_G \,\mathsf{U}\, \varphi_G \mid \varphi_G \,\mathsf{M}\, \varphi_G \\
\varphi_S &::= \varphi_B \mid \neg\varphi_G \mid \varphi_S \wedge \varphi_S \mid \varphi_S \vee \varphi_S \mid \varphi_G \to \varphi_S \mid \mathsf{X}\varphi_S \mid \mathsf{G}\varphi_S \mid \varphi_S \,\mathsf{R}\, \varphi_S \mid \varphi_S \,\mathsf{W}\, \varphi_S \\
\varphi_O &::= \varphi_G \mid \varphi_S \mid \neg\varphi_O \mid \varphi_O \wedge \varphi_O \mid \varphi_O \vee \varphi_O \mid \varphi_O \leftrightarrow \varphi_O \mid \varphi_O \oplus \varphi_O \mid \varphi_O \to \varphi_O \mid \mathsf{X}\varphi_O \\
&\quad \mid \varphi_O \,\mathsf{U}\, \varphi_G \mid \varphi_O \,\mathsf{R}\, \varphi_S \mid \varphi_S \,\mathsf{W}\, \varphi_O \mid \varphi_G \,\mathsf{M}\, \varphi_O
\end{aligned}$$

Formulas built using the rule for φ_S (resp. φ_G) are syntactic safety (resp. syntactic guarantee) and also correspond to the class Σ_1 (resp. Π_1) defined by Esparza et al. [27]. The "bottom" class, built using the φ_B rule, is the intersection of these two syntactic classes; it differs from Esparza's Δ_0 class in that it allows X operators. If we ignore the last four options in φ_O, the φ_O rule represents Boolean combinations of syntactic safety and syntactic guarantee, and differs from Esparza's Δ_1 class in that it is also closed under the X operator. The last four options in φ_O allow to capture more obligations syntactically [11,10].

Note that we never require formulas to be in *negative normal form* (NNF). Translations that require NNF suffer a blowup from the removal of operators $\leftrightarrow$ and $\oplus$. Instead, since DWA are closed under Boolean operations, we support these operators naturally.

In what follows, let $\mathsf{LTL}_O(\mathcal{P})$ designate the set of syntactic obligations that can be built over atomic propositions $\mathcal{P}$ and similarly for $\mathsf{LTL}_B(\mathcal{P})$, $\mathsf{LTL}_G(\mathcal{P})$, and $\mathsf{LTL}_S(\mathcal{P})$.

3.2 Translation to MTDWA

We are going to build an MTDWA $\langle Q, \mathcal{P}, \iota, \Delta, \lambda\rangle$, where states $Q \subseteq \mathsf{LTL}_O(\mathcal{P})$ are identified by syntactic obligations, and $\iota \in \mathsf{LTL}_O(\mathcal{P})$ is the formula we want to translate.

Computing successors using MTBDDs. Assuming $\boxed{\alpha}$ represents a terminal labeled by $\alpha \in \mathsf{LTL}_O(\mathcal{P})$, the MTBDD representation of the successors of a state, i.e., the function $\mathsf{tr} : \mathsf{LTL}_O(\mathcal{P}) \to \mathsf{MTBDD}(\mathcal{P}, \mathsf{LTL}_O(\mathcal{P}))$, is defined as follows:

$$\mathsf{tr}(tt) = \boxed{tt} \qquad \mathsf{tr}(ff) = \boxed{ff} \qquad \mathsf{tr}(a) = \overset{a}{\bigcirc}\!\!\begin{array}{l}\swarrow\,\boxed{ff}\\ \searrow\,\boxed{tt}\end{array} \quad \text{for any } a \in \mathcal{P}$$

$$\mathsf{tr}(\mathsf{X}\varphi) = \boxed{\varphi} \qquad \mathsf{tr}(\neg\varphi) = \neg\mathsf{tr}(\varphi)$$

$$\mathsf{tr}(\alpha \odot \beta) = \mathsf{tr}(\alpha) \odot \mathsf{tr}(\beta) \quad \text{for any Boolean operator } \odot \in \{\wedge, \vee, \to, \leftrightarrow, \oplus, ...\}$$

$$\mathsf{tr}(\alpha\,\mathsf{U}\,\beta) = \mathsf{tr}(\beta) \vee (\mathsf{tr}(\alpha) \wedge \boxed{\alpha\,\mathsf{U}\,\beta}) \qquad \mathsf{tr}(\alpha\,\mathsf{M}\,\beta) = \mathsf{tr}(\beta) \wedge (\mathsf{tr}(\alpha) \vee \boxed{\alpha\,\mathsf{M}\,\beta})$$

$$\mathsf{tr}(\alpha\,\mathsf{W}\,\beta) = \mathsf{tr}(\beta) \vee (\mathsf{tr}(\alpha) \wedge \boxed{\alpha\,\mathsf{W}\,\beta}) \qquad \mathsf{tr}(\alpha\,\mathsf{R}\,\beta) = \mathsf{tr}(\beta) \wedge (\mathsf{tr}(\alpha) \vee \boxed{\alpha\,\mathsf{R}\,\beta})$$

$$\mathsf{tr}(\mathsf{F}\varphi) = \mathsf{tr}(\varphi) \vee \boxed{\mathsf{F}\varphi} \qquad \mathsf{tr}(\mathsf{G}\varphi) = \mathsf{tr}(\varphi) \wedge \boxed{\mathsf{G}\varphi}$$

Deciding acceptance of states. In this simple LTL fragment, the acceptance of a state $\varphi \in \mathsf{LTL}_O(\mathcal{P})$ can be defined by simply considering the Boolean combination of safety and guarantee formulas that φ represents. This can be done using the function $\lambda(\varphi)$ defined below, which checks the top-level operator of each maximal temporal subformula of φ (strong operators F, U, M are rejecting; weak operators G, W, R are accepting) and builds their Boolean combinations.

$$\lambda(ff) = \lambda(\alpha\,\mathsf{U}\,\beta) = \lambda(\alpha\,\mathsf{M}\,\beta) = \lambda(\mathsf{F}\varphi) = \bot \qquad \lambda(tt) = \lambda(\alpha\,\mathsf{W}\,\beta) = \lambda(\alpha\,\mathsf{R}\,\beta) = \lambda(\mathsf{G}\varphi) = \top$$

$$\lambda(\alpha \odot \beta) = \lambda(\alpha) \odot \lambda(\beta) \quad \text{for any Boolean operator } \odot \in \{\wedge, \vee, \leftarrow, \leftrightarrow, \oplus, ...\}$$

$$\lambda(\neg\varphi) = \neg\lambda(\varphi) \qquad \lambda(\mathsf{X}\varphi) = \lambda(\varphi) \qquad \lambda(a) = * \text{ for any } a \in \mathcal{P}$$

Value $*$ is a wildcard that should be ignored during Boolean operations, as shown below.

$$\begin{array}{lllll}
\top \wedge * = \top & \bot \wedge * = \bot & * \wedge * = * & \top \oplus * = \top & \bot \oplus * = \top \\
\top \vee * = \top & \bot \vee * = \bot & * \vee * = * & * \oplus * = * & \neg * = * \\
\top \to * = \bot & \bot \to * = \top & * \to * = * & * \to \top = \top & * \to \bot = \bot \\
\top \leftrightarrow * = \top & \bot \leftrightarrow * = \top & * \leftrightarrow * = * & &
\end{array}$$

A state φ such that $\lambda(\varphi) = *$ is a Boolean combination of atomic propositions, possibly with X operators. As it cannot be part of a cycle, its acceptance can be arbitrary.

In our implementation tt and ff never occur as arguments of Boolean operators or of X: such formulas are automatically simplified (e.g., trying to construct $\mathsf{X}(tt) \wedge a$ will return a). Therefore, $\lambda(tt)$ and $\lambda(ff)$ are never called recursively.

Ensuring termination with propositional equivalence. It is tempting to define our automaton $\langle Q, \mathcal{P}, \iota, \Delta, \lambda\rangle$ by letting $\Delta = \mathsf{tr}$, and by setting Q to be equal to all the formulas that can be reached transitively from ι by applying tr. This does not work because the set Q constructed this way is not necessarily finite. The classical solution, which we reuse, is to use propositional equivalence [26,24]. Cf. App. C

In previous work on $\mathrm{LTL_f}$ translation [24], we used propositional equivalence to simplify formulas created at every step of the computation of $\mathsf{tr}(\cdot)$. In the current implementation we found that it was faster to do it as a separate pass: first compute

the MTBDD $m = \mathsf{tr}(\varphi)$, then replace all the leaves of m by a representative of their propositional-equivalence class. This way, we avoid computing propositional equivalence for the intermediate computations of $\mathsf{tr}(\cdot)$.

Additionally, as part of propositional equivalence we consider the distribution of the X operators through Boolean operators. E.g., we interpret $\mathsf{X}(\alpha \wedge \beta) \vee \mathsf{X}(\beta)$ as $(\mathsf{X}(\alpha) \wedge \mathsf{X}(\beta)) \vee \mathsf{X}(\beta)$, so it can be found to be equivalent to $\mathsf{X}(\beta)$.

From now on, we assume that tr includes propositional-equivalence, so that Q, the set of formulas reachable transitively from ι by applying tr, is guaranteed to be finite.

Correctness of the construction. While propositional equivalence ensures that our construction terminates, its correctness is established by the following lemma and theorem.

Cf. App. D

Lemma 1. *For any $\iota \in \mathsf{LTL}_O(\mathcal{P})$ let $D_\iota = \langle Q, \mathcal{P}, \iota, \mathsf{tr}, \lambda\rangle$ be the automaton constructed as described above, keeping $\lambda : Q \to \mathbb{B} \cup \{*\}$ (unlike in Definition 1). The following statements hold:*

1. *If a state $\varphi \in Q$ is such that $\lambda(\varphi) = *$, this state belongs to a trivial SCC of D_ι.*
2. *For any bottom formula $\iota \in \mathsf{LTL}_B(\mathcal{P})$, the only possible non-trivial SCCs of D_ι are the states tt and ff (at least one of these has to exist).*
3. *For any guarantee formula $\iota \in \mathsf{LTL}_G(\mathcal{P})$, all states $\varphi \in Q \setminus \{tt\}$ that belong to non-trivial SCCs of D_ι have $\lambda(\varphi) = \bot$.*
4. *For any safety formula $\iota \in \mathsf{LTL}_S(\mathcal{P})$, all states $\varphi \in Q \setminus \{ff\}$ that belong to non-trivial SCCs of D_ι have $\lambda(\varphi) = \top$.*
5. *For any obligation formula $\iota \in \mathsf{LTL}_O(\mathcal{P})$, all the states of any non-trivial SCC of D_ι get assigned the same value by λ, and this value is either $\top$ or $\bot$.*

Since the states φ for which $\lambda(\varphi) = *$ cannot be part of a cycle, their acceptance can be chosen arbitrarily. In Section 3.3, we show that this choice is constrained if we want to minimize the automaton. For now, in order to formalize this definition, we declare those states as rejecting using the function $\lambda' : Q \to \mathbb{B}$ defined as follows:

$$\lambda'(\varphi) = \begin{cases} \top & \text{if } \lambda(\varphi) = \top \\ \bot & \text{if } \lambda(\varphi) \in \{\bot, *\} \end{cases}$$

Theorem 1. *For every $\iota \in \mathsf{LTL}_O(\mathcal{P})$ the automaton $D_\iota = \langle Q, \mathcal{P}, \iota, \mathsf{tr}, \lambda'\rangle$ constructed above is an MTDWA, and it satisfies $L(\iota) = L(D_\iota)$.*

Proof. The fact that D_ι is weak is a corollary that follows directly from Lemma 1. The language $L(D_\iota)$ being equivalent to $L(\iota)$ follows by noticing that tr implements the local truth of LTL formulas and by induction on the structure of formulas. □

3.3 Minimizing MTDWA

Contrary to DBAs [66], DWAs can be minimized in polynomial time [46]. In fact, after a very cheap preprocessing to decide for each transient state (i.e., a state that cannot be part of a loop) whether it should be accepting or rejecting, a DWA can be minimized

as if it were a DFA. Löding's preprocessing [46] is based on a simple ranking function over the strongly connected components (SCC) of the DWA, such that the parity of the rank indicates the acceptance of the SCC. We can perform an SCC decomposition over the MTDWA directly, by simply interpreting the forest of MTBDDs (which is normally acyclic) as a graph where a leaf $\boxed{\alpha}$ is connected to the root of $\Delta(\alpha)$.

The minimization itself is easily done over MTBDDs by using a variant of Moore's partition-refinement algorithm [55] similar to what is implemented in Mona [35]. Assuming that each state labeled by α of the MTDWA is assigned to block $B(\alpha) \in \mathbb{N}$ in the partition of the current iteration, the next partition can be found by creating for each state s a temporary $\Delta'(s) \in \mathsf{MTBDD}(\mathcal{P}, \mathbb{N})$ in which all leaves $\boxed{\alpha}$ of $\Delta(s)$ are replaced by $\boxed{B(\alpha)}$, and partitioning the set of states according to the different MTBDDs in Δ'.

An optimization not mentioned in Löding's paper is that the only states that can be merged during minimization are those that share the same rank. Consequently, Löding's ranking function can be used to define the initial partition of the minimization.

4 LTL_O Reactive Synthesis

A reactive controller can be thought of as an electronic circuit that produces output signals based on a history of input signals. LTL Reactive Synthesis is the problem of building such a controller (usually as a Mealy machine) such that the combined histories of input and output signals satisfy some given LTL specification.

We implement synthesis to AIGER circuits [6,65], but for lack of space we only discuss the "realizability" problem, defined as follows.

Definition 2 ([42,29]). *Given two disjoint sets of variables $\mathcal{I}$ (inputs) and $\mathcal{O}$ (outputs), a controller is a function $\rho : (\mathbb{B}^{\mathcal{I}})^* \to \mathbb{B}^{\mathcal{O}}$ that, given a history of assignments of input variables, produces an assignment of output variables.*

Given a word of input assignments $\sigma \in (\mathbb{B}^{\mathcal{I}})^\omega$, the controller can be used to generate a word of output assignments $\sigma_\rho \in (\mathbb{B}^{\mathcal{O}})^\omega$. The definition of σ_ρ may use two semantics depending on whether we want the controller to have access to the current input assignment to decide the output assignment:
Mealy semantics: $\sigma_\rho(i) = \rho(\sigma(..i))$ *for all* $i \geq 0$.
Moore semantics: $\sigma_\rho(i) = \rho(\sigma(..i-1))$ *for all* $i \geq 0$.

A formula $\varphi \in \mathsf{LTL}(\mathcal{I} \cup \mathcal{O})$ is said to be Mealy-realizable*, or* Moore-realizable*, if there exists a controller ρ such that for every word $\sigma \in (\mathbb{B}^{\mathcal{I}})^\omega$, it holds that $(\sigma \sqcup \sigma_\rho) \in \mathcal{L}(\varphi)$ using the desired semantics.*

Formula φ_1 (from Figure 1) is both Mealy-realizable and Moore-realizable.

For general LTL, synthesis or realizability can be reduced to the translation of the specification into a deterministic parity automaton, which is then interpreted and solved as a two-player game with parity acceptance: one player plays the input signals, the second player plays the output signals. The specification is realizable if the output player has a strategy to ensure that the parity acceptance is satisfied.

For syntactic obligations, realizability and synthesis are a lot easier: any LTL_O formula can be converted to a DWA, which can also be interpreted as a two-player game with weak acceptance. Such games are known to be solvable in linear time [1, Section

6.1]. They can also be reduced to the emptiness check of 1-letter weak alternating automata [43, Theorem 4.7], or seen as a special case of weak parity games [12]. These three algorithms work similarly. Since the automaton is weak, we know that the cycles in one SCC are either all accepting (winning for the output player) or all rejecting (winning for the input player), but except for SCCs without successors, it might be possible for players to escape an SCC that is losing for them. The game is therefore solved by enumerating the SCCs and processing them bottom-up: terminal SCCs can be determined as winning for one player or the other according to their acceptance. Then the attractor of these states is grown by backpropagation. Moving up in the SCCs, undetermined states can be determined according to the acceptance of the SCC.

This algorithm can be implemented directly on the MTDWA structure, by interpreting it as a game where the input (resp., output) player makes the decision for the input (resp., output) variables, and where a play that reaches $\boxed{\alpha}$ continues from the root node associated with α. For Mealy semantics, input variables should be ordered to appear before output variables; for Moore semantics, it should be the opposite.

Cf. App. E

We implement this game solving over MTDWA on-the-fly, together with the automaton construction of Section 3. The setup is very similar to our previous solution for LTL_f synthesis [24] where we had to construct a DFA represented using MTBDDs, and interpret it as a reachability game to be solved on-the-fly. In a reachability game, the exploration order is not important: this previous work used a BFS just because we found it marginally better than DFS. Here, for obligation synthesis, we have to explore the states in a topological order of their SCCs. Enumerating SCCs during the on-the-fly construction of an automaton is a well studied topic, and is typically used by on-the-fly explicit LTL model checkers [34,32,62]. These algorithms are all based on a DFS exploration of the automaton. Our implementation uses Dijkstra's SCC algorithm [19,20], and we construct the MTDWA as the DFS progresses. The implementation reuses the linear backpropagation over incomplete graphs from previous work [24, Algorithm 1].

5 Implementation and Evaluation

Cf. App. F

The new constructions for translation, minimization, and game solving have been implemented [22] in Spot [23] and released as version 2.15. Many graphical examples of these constructions can be found at https://spot.lre.epita.fr/ipynb/mtdswa.html Below, we evaluate how the translation and synthesis improved compared to the previous version. We also include a comparison to a few third-party tools for reference.

5.1 Translation

The new translation procedure is used by Spot's `translator` class whenever the input formula is a syntactic obligation and the user requests a deterministic output. For evaluation purposes, we use Spot's `ltl2tgba` command-line tool, which converts LTL into various kinds of automata and outputs them in the HOA format [2].

Figure 2 shows four translation pipelines that `ltl2tgba` can use to translate syntactic obligations into DWAs. The dashed rectangle corresponds to the aforementioned `translator` class: it translates an LTL formula into an explicit ω-automaton, but it

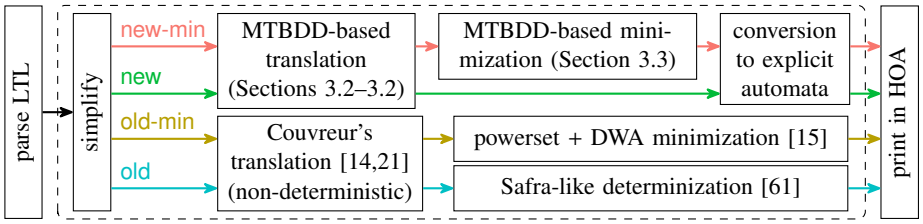

Fig. 2: Four pipelines usable by `ltl2tgba` to translate a syntactic obligation: new-min, new, old-min, and old. The `spot::translator` class implements the dashed box.

can be configured to use different approaches. The path old-min corresponds to the default behavior of Spot 2.14: after some simple syntactic simplifications, the formula is converted into a non-deterministic Büchi automaton using Couvreur's translation algorithm [14,21], to obtain a weak NBA; the result is then determinized by powerset construction and minimized as described by Dax et al. [15], and finally the resulting automaton is output in HOA. The old pipeline is used when the DWA-minimization is explicitly disabled or not applicable: this path works for any input formula, not just obligations. In that case, since the output of the translation is non-deterministic, but the goal is to have a deterministic automaton, the automaton is determinized using a Safra-based construction [61]. Cf. App. B.

In both old-min and old, automata are represented as explicit graphs with edges labeled by Boolean formulas over propositions (represented as BDDs) [21,23], as on the left-hand side of Figure 1. In the new pipelines, called new-min and new, the MTBDD-based translation described in Section 3 produces an MTDWA as on the right-hand side of Figure 1, and that can optionally be minimized using the same data structure. In order to output an MTDWA in the HOA format, we first convert this MTBDD-based representation into an explicit representation. This conversion has a significant cost because it has to enumerate all paths in the MTBDDs. Nonetheless, we will see that these new pipelines are still competitive, despite the costly conversion.

As far as we know, Spot is the only tool that can translate syntactic obligations into DWA that are guaranteed to be minimal (via the old-min or new-min pipelines). We can, however, compare to tools that produce deterministic ω-automata even if they do not guarantee minimality. We consider ltl3tela 2.1.1 [49] and ltl2dela from Owl 21.0 [40]. (The latter is an evolution of Delag [56], which we did not run.)

For a fair comparison, we set up all tools to read an LTL formula and produce an equivalent deterministic ω-automaton (with any acceptance condition) in the HOA format. We request complete automata from all tools except ltl3tela (which lacks this option). We measure the entire execution of each tool using `ltlcross` with a timeout of 60 seconds. The experiment was run one task at a time on a dedicated Core i7-3770 with *Turbo Boost* disabled, and the frequency scaled down to 1.6GHz to prevent CPU throttling.

We evaluated the translation pipelines on 494 syntactic obligations: Details in App. G

- 310 formulas representing instances of 17 scalable patterns [33,13,33,41,68];
- 76 formulas collected from various works [25,67,28,57,36];

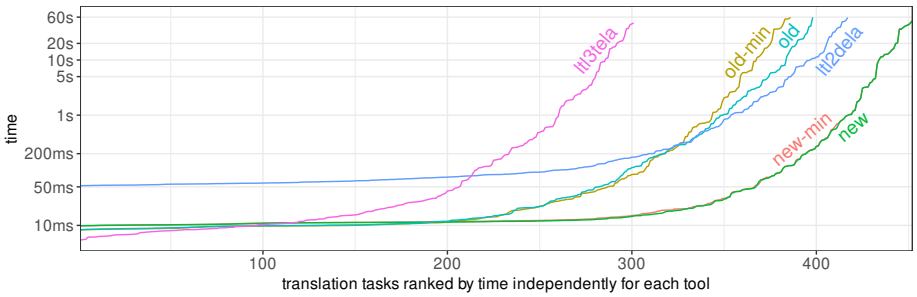

Fig. 3: Comparison of translation tools on 494 syntactic obligations from the literature.

– 108 unique formulas from the synthesis competition [37]: all the instances that are syntactic obligations, except for two formulas syntactically equivalent to *tt* or *ff*.

Figure 3, where the time axis is logarithmic, clearly shows how Spot's translation has improved over this set of formulas. Note that new and new-min are barely distinguishable, suggesting that the guarantee to build minimal automata comes at no extra cost.

Keep in mind, however, that we measured the runtime of a complete translation process, including the conversion to explicit automata (for new and new-min) and HOA output (for all tools). We now turn to the evaluation of how the MTBDD-based representation of DWA improves synthesis: in this case, we do not need to convert automata to an explicit representation, saving a lot of time.

5.2 Synthesis

Cf. App. I

Previous versions of Spot's `ltlsynt` converted an LTL specification by translating it to a deterministic parity automaton (DPA), turning this DPA into a two-player game, and solving the game [65]. In the case of syntactic obligations, the DPA was obtained by the old-min translation pipeline in Figure 2. This default behavior of `ltlsynt` 2.14 is what is called (old trans.) in this section.

The improvements described in this paper allow us to define two better configurations of `ltlsynt`. (new trans.) designates a configuration where the `spot::translator` class, used to produce the DPA, is changed to use the new-min translation pipeline in Figure 2, where the automaton is still converted to an explicit representation, to be solved as a game. The (new synt.) configuration corresponds to the new default of `ltlsynt`, where syntactic obligations are converted into an MTBDD-based representation, and the game is solved directly on that representation as described in this paper.

See App. J for memory usage.
See App. K about SemML.

Figure 4 shows the improved runtimes as a cactus plot. For this experiment, we used BenchExec 3.22 [5] to track time and memory usage. We include Strix 21.0 [48] (the winner of SyntComp'23) and SemML [39] (the winner of SyntComp'24) for comparison. (The winner of SyntComp'25 was `ltlsynt` due to a technical defect in SemML.) Moreover, SSyft [3], patched to use the Mealy semantics, is included as an example of a tool that is dedicated to syntactic-safety formulas: there are 48 of those in this benchmark. All tools were configured to check the specification for realizability: this way we do not include any time spent generating and outputting controllers.

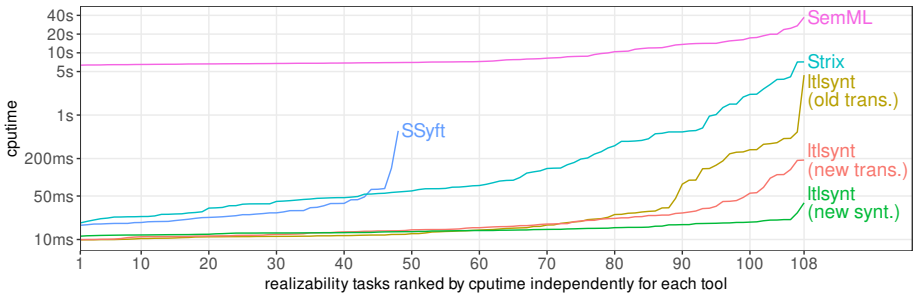

Fig. 4: Comparison of the three configurations of `ltlsynt` over the 108 specifications from SyntComp that are syntactic obligations. Third-party tools Strix, SemML, and SSyft are included for reference.

6 Conclusion

We have shown that the MTBDD-based translation has made `ltl2tgba` faster at converting syntactic obligations to (weak) DBAs. This holds even when we switch back to an explicit representation. In the case of synthesis, the latter switch back is not required and the weak game can be solved directly on the MTBDD-based representation, providing an even greater speedup.

The techniques described target syntactic obligations, so we focused our experiments on those. We expect, however, that those improvements will impact other types of specifications as well. Firstly, when a specification can be seen as a conjunction of output-disjoint sub-specifications, `ltlsynt` solves each sub-specification independently [30,65]. Therefore, even if the full specification is not a syntactic obligation, it could contain a sub-specification that is a syntactic obligation, and that will be solved using the improved MTBDD-based technique. Secondly, a possible way to obtain a DPA is to first translate the specification into a deterministic Emerson-Lei automaton (DELA), which can then be converted into a DPA [63,9]. To obtain a DELA, Spot generalizes the compositional translation used by the `Delag` tool [56]: the formula to translate is broken into top-level Boolean operators, subformulas are translated to deterministic automata according to their nature, and those automata are then recombined. In this approach, subformulas that are syntactic obligations will now be translated much more efficiently. See App. L.

Obvious future work would be to generalize the MTBDD-based translation, so that all LTL formulas can be turned into MTBDD-based DELA, probably using Δ_2-normalization [27], converting top-level temporal formulas to deterministic Büchi and co-Büchi automata, and using Boolean combinations for the final automaton.

Data Availability Statement Code and benchmark data are archived on Zenodo [22].

Acknowledgments. NP is supported by Swedish Research Council (VR) project (No. 2020-04963) and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. GDG is partially supported by the EPSRC grant EP/Y028872/1, Mathematical Foundations of Intelligence: An Erlangen Programme for AI.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Amram, G., Bansal, S., Fried, D., Tabajara, L.M., Vardi, M.Y., Weiss, G.: Adapting behaviors via reactive synthesis. In: Proceedings of the 33rd International Conference on Computer Aided Verification (CAV'21). pp. 870–89. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-81685-8_41
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi Omega-Automata format. In: Proceedings of the 27th International Conference on Computer Aided Verification (CAV'15). Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (Jul 2015). https://doi.org/10.1007/978-3-319-21690-4_31
3. Bansal, S., De Giacomo, G., Di Stasio, A., Li, Y., Vardi, M.Y., Zhu, S.: Compositional safety LTL synthesis. In: Lal, A., Tonetta, S. (eds.) Proceedings of the 14th International Conference on Verified Software, Theories, Tools and Experiments (VSTTE'22). pp. 1–19. Springer International Publishing (2023). https://doi.org/10.1007/978-3-031-25803-9_1
4. Bansal, S., Li, Y., Tabajara, L.M., Vardi, M.Y.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: Proceedings of the 34th national conference on Artificial intelligence (AAAI'20). pp. 9766–9774. AAAI Press (2020). https://doi.org/10.1609/AAAI.V34I06.6528
5. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer **21**, 1–29 (Feb 2019). https://doi.org/10.1007/s10009-017-0469-y
6. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011), https://fmv.jku.at/aiger/
7. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences **78**(3), 911–938 (2012). https://doi.org/10.1016/J.JCSS.2011.08.007
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (Aug 1986). https://doi.org/10.1109/TC.1986.1676819
9. Casares, A., Duret-Lutz, A., Meyer, K.J., Renkin, F., Sickert, S.: 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 (TACAS'22). Lecture Notes in Computer Science, vol. 13244, pp. 99–117 (Apr 2022). https://doi.org/10.1007/978-3-030-99527-0_6
10. Černá, I., Pelánek, R.: 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, pp. 318–327. Springer-Verlag, Bratislava, Slovak Republic (Aug 2003). https://doi.org/10.1007/978-3-540-45138-9_26
11. Chang, E.Y., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92). pp. 474–486. Springer-Verlag, London, UK (1992). https://doi.org/10.1007/3-540-55719-9_97
12. Chatterjee, K.: Linear time algorithm for weak parity games. Tech. Rep. UCB/EECS-2006-153, Electrical Engineering and Computer Sciences, University of California at Berkeley (Nov 2008), https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-153.html

13. Cichoń, J., Czubak, A., Jasiński, A.: Minimal Büchi automata for certain classes of LTL formulas. In: Proceedings of the Fourth International Conference on Dependability of Computer Systems (DepCoS'09). pp. 17–24. IEEE Computer Society (2009). https://doi.org/10.1109/DepCoS-RELCOMEX.2009.31
14. Couvreur, J.M.: On-the-fly verification of temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99). Lecture Notes in Computer Science, vol. 1708, pp. 253–271. Springer-Verlag (Sep 1999). https://doi.org/10.1007/3-540-48119-2_16
15. Dax, C., Eisinger, J., Klaedtke, F.: Mechanizing the powerset construction for restricted classes of ω-automata. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07). Lecture Notes in Computer Science, vol. 4762. Springer-Verlag (Oct 2007). https://doi.org/10.1007/978-3-540-75596-8_17
16. De Giacomo, G., Favorito, M.: Compositional approach to translate LTL_f/LDL_f into deterministic finite automata. In: Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21). pp. 122–130 (2021). https://doi.org/10.1609/icaps.v31i1.15954
17. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13). pp. 854–860. AAAI Press (Aug 2013). https://doi.org/10.5555/2540128.2540252
18. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15). pp. 1558–1564. AAAI Press (2015). https://doi.org/10.5555/2832415.2832466
19. Dijkstra, E.W.: EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF (May 1973)
20. Dijkstra, E.W.: Finding the maximal strong components in a directed graph. In: A Discipline of Programming, chap. 25, pp. 192–200. Prentice-Hall (1976)
21. Duret-Lutz, A.: LTL translation improvements in Spot 1.0. International Journal on Critical Computer-Based Systems **5**(1/2), 31–54 (Mar 2014). https://doi.org/10.1504/IJCCBS.2014.059594
22. Duret-Lutz, A.: Supporting material for "Fast Obligation Translation and Synthesis" (Jan 2026). https://doi.org/10.5281/zenodo.19812382
23. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Proceedings of the 34th International Conferencermane on Computer Aided Verification (CAV'22). Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (Aug 2022). https://doi.org/10.1007/978-3-031-13188-2_9
24. Duret-Lutz, A., Zhu, S., Piterman, N., De Giacomo, G., Vardi, M.Y.: Engineering an LTLf synthesis tool. In: Proceedings of the 29th International Conference on Implementation and Applications of Automata (CIAA'25). Lecture Notes in Computer Science, vol. 15981, pp. 129–147. Springer (Sep 2025). https://doi.org/10.1007/978-3-032-02602-6_10
25. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis, M. (ed.) Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98). pp. 7–15. ACM Press (Mar 1998). https://doi.org/10.1145/298595.298598
26. Esparza, J., Křetínský, J., Sickert, S.: 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). pp. 384–393. ACM (2018). https://doi.org/10.1145/3209108.3209161
27. Esparza, J., Rubio, R., Sickert, S.: Efficient normalization of linear temporal logic. Journal of the ACM **71**(2) (Apr 2024). https://doi.org/10.1145/3651152

28. Etessami, K., Holzmann, G.J.: 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, pp. 153–167. Springer-Verlag, Pennsylvania, USA (2000). https://doi.org/10.1007/3-540-44618-4_13
29. Finkbeiner, B.: Synthesis of reactive systems. In: Javier Esparza, Orna Grumberg, S.S. (ed.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series — D: Information and Communication Security, vol. 45, pp. 72–98. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-72
30. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: Proceedings for the 13th NASA Formal Methods Symposium (NFM'21). Lecture Notes in Computer Science, vol. 12673, pp. 113–130. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_8
31. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design **10**(2/3), 149–169 (1997). https://doi.org/10.1023/A:1008647823331
32. Gaiser, A., Schwoon, S.: Comparison of algorithms for checking emptiness on Büchi automata. In: Hlinený, P., Matyás, V., Vojnar, T. (eds.) Procedings of Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09). OASICS, vol. 13. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany (Nov 2009). https://doi.org/10.4230/DROPS.MEMICS.2009.2349
33. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: Proceedings of the 13th International SPIN Workshop (SPIN'06). Lecture Notes in Computer Science, vol. 3925, pp. 53–70. Springer (2006). https://doi.org/10.1007/11691617_4
34. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science **345**(1), 60–82 (Nov 2005). https://doi.org/10.1016/j.tcs.2005.07.004
35. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95). pp. 89–110. Springer Berlin Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5
36. Holeček, J., Kratochvíla, T., Řehák, V., Šafránek, D., Šimeček, P.: Verification results in Liberouter project. Tech. Rep. 03, CESNET (September 2004), http://archiv.cesnet.cz/doc/techzpravy/2004/verificationresults/
37. Jacobs, S., Perez, G.A., Abraham, R., Bruyère, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K.J., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018–2021. arXiV (Jun 2022). https://doi.org/10.48550/ARXIV.2206.00251
38. Klarlund, N., Møller, A.: MONA version 1.4, user manual. Tech. rep., BRICS (Jul 2001), https://www.brics.dk/mona/mona14.pdf
39. Křetínský, J., Meggendorfer, T., Prokop, M., Zarkhah, A.: SemML: Enhancing automata-theoretic LTL synthesis with machine learning. In: Gurfinkel, A., Heule, M. (eds.) Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'25). pp. 233–253. Springer Nature Switzerland (2025). https://doi.org/10.1007/978-3-031-90643-5_12
40. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA'18). Lecture Notes in Computer Science, vol. 11138, pp. 543–550. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_34

41. Kupferman, O., Rosenberg, A.: The blow-up in translating ltl to deterministic automata. In: van der Meyden, R., Smaus, J.G. (eds.) Proceedings of the 6th International Workshop on Model Checking and Artificial Intelligence (MochArt'11). pp. 85–94. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20674-0_6
42. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05). pp. 531–542 (2005). https://doi.org/10.1109/SFCS.2005.66
43. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM **47**(2), 312–360 (Mar 2000). https://doi.org/10.1145/333979.333987
44. Li, Y., Xiao, S., Zhu, S., Li, J., Pu, G.: A compositional framework for on-the-fly LTL$_f$ synthesis. In: Proceedings of the 28th European Conference on Artificial Intelligence (ECAI'25). pp. 1711–1718. Frontiers in Artificial Intelligence and Applications (2025). https://doi.org/10.3233/FAIA250999
45. Löding, C.: Methods for the Transformation of ω-Automata: Complexity and Connection to Second Order Logic. Diploma thesis, Institute of Computer Science and Applied Mathematics Christian-Albrechts-University of Kiel (1998), https://www.lics.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaabcqdty
46. Löding, C.: Efficient minimization of deterministic weak ω-automata. Information Processing Letters **79**(3), 105–109 (2001). https://doi.org/10.1016/S0020-0190(00)00183-6
47. Long, D.: BDD library. source archive, https://www.cs.cmu.edu/~modelcheck/bdd.html
48. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020). https://doi.org/10.1007/S00236-019-00349-3
49. Major, J., Blahoudek, F., Strejcek, J., Sasaráková, M., Zboncáková, T.: 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, pp. 357–365 (2019). https://doi.org/10.1007/978-3-030-31784-3_21
50. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC'90). pp. 377–410. ACM, New York, NY, USA (1990). https://doi.org/10.1145/93385.93442
51. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: Specification. Springer (1992). https://doi.org/10.1007/978-1-4612-0931-7
52. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Safety. Springer (1995). https://doi.org/10.1007/978-1-4612-4222-2
53. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Response. In: Manna, Z., Peled, D.A. (eds.) Time for Verification: Essays in Memory of Amir Pnueli, Lecture Notes in Computer Science, vol. 6200, pp. 279–361. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_13
54. Minato, S.i.: Representation of Multi-Valued Functions, pp. 39–47. Springer US, Boston, MA (1996). https://doi.org/10.1007/978-1-4613-1303-8_4
55. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata studies, pp. 129–153. No. 34 in Annals of Mathematical Studies, Princeton University Press (1956)
56. Müller, D., Sickert, S.: LTL to deterministic Emerson-Lei automata. In: Proceedings of the 8th International Sumposium on Games, Automata, Logics and Formal Verification (GandALF'17). Electronic Proceedings in Theoretical Computer Science, vol. 256, pp. 180–194. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.256.13
57. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Proceedings of the 14th international SPIN conference on Model checking software (Spin'07). Lecture Notes in

Computer Science, vol. 4595, pp. 263–267. Springer-Verlag (2007). https://doi.org/10.1007/978-3-540-73370-6_17
58. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06). Lecture Notes in Computer Science, vol. 3855, pp. 364–380. Springer (2006). https://doi.org/10.1007/11609773_24
59. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77). pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
60. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'89). Association for Computing Machinery (1989). https://doi.org/10.1145/75277.75293
61. Redziejowski, R.: An improved construction of deterministic omega-automaton using derivatives. Fundamenta Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744
62. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Three SCC-based emptiness checks for generalized Büchi automata. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19). Lecture Notes in Computer Science, vol. 8312, pp. 668–682. Springer (Dec 2013). https://doi.org/10.1007/978-3-642-45221-5_44
63. Renkin, F., Duret-Lutz, A., Pommellet, A.: 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, pp. 127–143. Springer (Oct 2020). https://doi.org/10.1007/978-3-030-59152-6_7
64. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Effective reductions of Mealy machines. In: Proceedings of the 42nd International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'22). Lecture Notes in Computer Science, vol. 13273, pp. 170–187. Springer (Jun 2022). https://doi.org/10.1007/978-3-031-08679-3_8
65. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting `ltlsynt`. Formal Methods in System Design (2023). https://doi.org/10.1007/s10703-022-00407-6
66. Schewe, S.: Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete
67. Somenzi, F., Bloem, R.: Efficient Büchi automata for LTL formulae. In: Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00). Lecture Notes in Computer Science, vol. 1855, pp. 247–263. Springer-Verlag, Chicago, Illinois, USA (2000). https://doi.org/10.1007/10722167_21
68. Tabakov, D., Vardi, M.Y.: Optimized temporal monitors for SystemC. In: Proceedings of the 1st International Conference on Runtime Verification (RV'10). Lecture Notes in Computer Science, vol. 6418, pp. 436–451. Springer (Nov 2010). https://doi.org/10.1007/978-3-642-16612-9_33
69. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science (SFCS'85). pp. 327–338. IEEE (1985). https://doi.org/10.1109/SFCS.1985.12
70. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17). pp. 1362–1369 (2017). https://doi.org/10.24963/ijcai.2017/189

source	set size	property classes					syntactic classes				
		O	$O\backslash S\backslash G$	$S\backslash B$	$G\backslash B$	B	O	$O\backslash S\backslash G$	$S\backslash B$	$G\backslash B$	B
SyntComp [37]	959	≥262	≥172	≥73	≥7	≥7	110	59	48	1	2
Dwyer et al. [25]	55	40	2	37	1		25	13	11	1	
Somenzi&Bloem [67]	27	16	2	6	6	2	13	4	4	5	
Etessami&Holzmann [28]	12	5			4	1	5			5	
BEEM [57]	20	10	1	6	1	2	7	2	4	1	
Liberouter [36]	55	30		27	1	2	26	1	23	2	

Table 1: Various collections of LTL formulas classified in the lower-fragments of the temporal hierarchy, and in their syntactic fragments. Empty cells are 0.

These appendices and the margin notes that point to them are for the interested reviewers, but are not meant to be part of the final version of the article.

A Obligation Frequency

Table 1 shows the frequency of obligation formulas in various sets of LTL formulas. For each set, the column "size" is the number of formulas in the set, and the remaining columns show how many formulas fall into the various lower classes of the temporal hierarchy, or in their syntactical fragments.

The letters O, S, G, B refer respectively to obligations, safety, guarantee, and bottom classes, where "bottom" is the intersection of S and G. Since O includes S and G and both include B, we report instead on the difference between these sets. E.g. $O\backslash S\backslash G$ counts obligation properties that are neither safety nor guarantee properties.

The "property class" columns were obtained by running the obligation detection algorithm presented in Appendix B. This classification is incomplete on the formulas from the Synthesis Competition because some of these formulas are too big to be processed by Spot's classifier. The lower bounds provided were computed with a timeout of 60 seconds. Out of the 959 formulas, 262 (27.3%) were classified as obligations, 401 (41.8%) as not obligation (high classes in the hierarchy), and the remaining 296 (30.9%) formulas could not be classified either because the classification would take more than 60 seconds, or because of other errors from Spot (e.g., a translation requiring more than 32 acceptance sets).

The "syntactic class" columns were obtained by checking whether the syntax tree of each formula follow the grammar rules from Section 3.1.

For instance, in the 27 formulas collected by Somenzi&Bloem [67], 16 represent obligation properties. In these 16 properties, 2 are obligations that are neither safety nor guarantee, 6 are safety properties that are not guarantee properties, 6 are guarantee properties that are not safety properties, and 2 are both guarantee and safety.[7] If we consider only the syntactic characterization of these classes, 13 formulas are syntactic

[7] The two formulas from Somenzi&Bloem [67] that are in B are: "$(\mathsf{X}p_0 \mathsf{U} \mathsf{X}p_1) \vee \neg\mathsf{X}(p_0 \mathsf{U} p_1)$" and "$(\mathsf{X}p_0 \mathsf{U} p_1) \vee \neg\mathsf{X}(p_0 \mathsf{U} (p_0 \wedge p_1))$". They belong to B because they are both tautologies. Syntactically, however, the lowest fragment they belong to is O.

obligations, 4 are syntactic obligations that are neither syntactic-safety nor syntactic-guarantee formulas, 4 are syntactic-safety formulas that are not syntactic-guarantee formulas, 4 are syntactic-guarantees that are not syntactic-safety, and 0 formulas are both syntactic-safety and syntactic-guarantee.

We can see in that table that obligations represent a frequently used class of formulas, and that, although the syntactic obligations do not capture all obligations, they represent a large portion of those.

B Deciding Membership to the Classes of the Temporal Hierarchy

To play and get familiar with Manna & Pnueli's hierarchy, the reader is invited to go to https://spot.lre.epita.fr/app/, click on the "STUDY" tab, and enter any LTL formula. That should display the class this formula belongs to, among other things.

For instance, the formula "`(a xor b) R (a U b)`" will be detected as an obligation property even if it is not a syntactic obligation. One equivalent syntactic obligation is the formula "`(a xor b) M (a U b) | G(a & b)`".

The classification performed on that page is done by Spot. To detect obligation properties, Spot implements a variation on the technique of Dax et al. [15, Section 3.2]. Our implementation improves upon the mentioned technique in the way it finds accepting SCCs, and by using only a single inclusion check at the end. Given some input formula φ for which we would like to know if it is an obligation, Spot proceeds as follows:

1. translate φ into a non-deterministic Büchi automaton N_φ,
2. ignoring the acceptance of states, use the powerset construction on N_φ to obtain the deterministic *structure* D_φ,
3. any SCC of D_φ that intersects an accepting SCC of N_φ in the synchronous product of $N_\varphi \otimes D_\varphi$ should be marked in D_φ as fully accepting
4. at this point D_φ is a DWA such that $\mathscr{L}(\varphi) \subseteq \mathscr{L}(D_\varphi)$ (because its accepting SCCs possibly capture runs that were not accepted by N_φ).
5. test $\mathscr{L}(D_\varphi) \subseteq \mathscr{L}(\varphi)$ by constructing a non-deterministic Büchi automaton $N_{\neg\varphi}$ for the negation of φ and then ensuring that the product $N_{\neg\varphi} \otimes D_\varphi$ is empty.

The success of the last inclusion check implies that $\mathscr{L}(D_\varphi) = \mathscr{L}(\varphi)$. Since D_φ is a DWA, this in turn implies that φ is an obligation. After minimizing this DWA, we can also detect guarantee properties (the only accepting state is an accepting sink) or safety properties (the only rejecting state is a rejecting sink).

As the size of N_φ could be exponential in $|\varphi|$, the size of D_φ is, in the worst case, doubly exponential. It follows that the algorithm above can be implemented in exponential space if D_φ is constructed on the fly.

Detection of higher classes in the hierarchy uses other techniques that are out of the scope of this paper.

C Why Propositional Equivalence is Needed

Consider $\varphi_1 = (\mathsf{G}a)\,\mathsf{W}\,(\mathsf{G}b)$. Following the branch $a \wedge b$ in the MTBDD $\mathsf{tr}(\varphi_1)$ leads to a leaf labeled by $\varphi_2 = (\mathsf{G}b) \vee ((\mathsf{G}a) \wedge ((\mathsf{G}a)\,\mathsf{W}\,(\mathsf{G}b)))$. Following $a \wedge b$ in the MTBDD

$\mathsf{tr}(\varphi_2)$ leads to a leaf labeled by $\varphi_3 = (\mathsf{G}b) \vee ((\mathsf{G}a) \wedge ((\mathsf{G}b) \vee ((\mathsf{G}a) \wedge ((\mathsf{G}a)\,\mathsf{W}\,(\mathsf{G}b)))))$, etc. Those formulas will keep growing.

Propositional equivalence can be defined as follows:

Definition 3 (Propositional Equivalence [26]). *For $\varphi \in \mathsf{LTL}(\mathcal{P})$, let φ_P be the Boolean formula obtained from φ by replacing every maximal temporal subformula ψ by a Boolean variable x_ψ. Two formulas $\alpha, \beta \in \mathsf{LTL}(\mathcal{P})$ are* propositionally equivalent*, denoted $\alpha \equiv \beta$, if α_P and β_P are equivalent Boolean formulas.*

If we compute the Boolean formulas φ_{1p}, φ_{2p}, φ_{3p} obtained by replacing $\mathsf{G}a$, $\mathsf{G}b$, and $(\mathsf{G}a)\,\mathsf{W}\,(\mathsf{G}b)$ by three different Boolean variables, and if we represent φ_{1p}, φ_{2p}, φ_{3p} as BDDs, we can see right away that $\varphi_{2p} = \varphi_{3p}$. Therefore φ_2 and φ_3 are propositionally equivalent, and we can replace the latter by the former, keeping the set of reachable formulas finite in the translation.

D Proof of Lemma 1

Lemma 1. *For any $\iota \in \mathsf{LTL}_O(\mathcal{P})$ let $D_\iota = \langle Q, \mathcal{P}, \iota, \mathsf{tr}, \lambda \rangle$ be the automaton constructed as described above, keeping $\lambda : Q \to \mathbb{B} \cup \{*\}$ (unlike in Definition 1). The following statements hold:*

1. *If a state $\varphi \in Q$ is such that $\lambda(\varphi) = *$, this state belongs to a trivial SCC of D_ι.*
2. *For any bottom formula $\iota \in \mathsf{LTL}_B(\mathcal{P})$, the only possible non-trivial SCCs of D_ι are the states tt and ff (at least one of these has to exist).*
3. *For any guarantee formula $\iota \in \mathsf{LTL}_G(\mathcal{P})$, all states $\varphi \in Q \setminus \{tt\}$ that belong to non-trivial SCCs of D_ι have $\lambda(\varphi) = \bot$.*
4. *For any safety formula $\iota \in \mathsf{LTL}_S(\mathcal{P})$, all states $\varphi \in Q \setminus \{ff\}$ that belong to non-trivial SCCs of D_ι have $\lambda(\varphi) = \top$.*
5. *For any obligation formula $\iota \in \mathsf{LTL}_O(\mathcal{P})$, all the states of any non-trivial SCC of D_ι get assigned the same value by λ, and this value is either $\top$ or $\bot$.*

Proof.

1. By definition of λ, a state $\varphi \in Q$ such that $\lambda(\varphi) = *$ is necessarily a Boolean combination of atomic propositions, with the possible addition of X operators. By the definition of tr, if φ is a purely Boolean combination of atomic propositions (i.e., no X), its successors can only be tt or ff. Additionally, if φ contains n nested X, its successors will have at most $n - 1$ nested X. Therefore φ cannot be part of a cycle.

The remaining points are shown by induction on the structure of the formula, following the fragments defined in Section 3.1.

2. Follows immediately from the previous point. The only formulas of $\mathsf{LTL}_B(\mathcal{P})$ for which λ does not return $*$ are ff and tt, and the automaton constructed for $\iota \in \mathsf{LTL}_B(\mathcal{P})$ will necessarily reach one (or both) of those states.

3. According to the definition of $\mathsf{tr}(\cdot)$, the only way to have a non-trivial SCC is by using the operators U and M. It follows that in every state ψ that appears in a non-trivial SCC we have $\lambda(\psi) = \bot$. Otherwise, the only accepting non-trivial SCC is *tt*, for which $\lambda(tt) = \top$.
   In the case of $\iota = \neg\varphi_S$ for $\varphi_S \in \mathsf{LTL}_S(\mathcal{P})$, the Lemma follows by duality and induction on the structure of the formula.
4. This is the dual of the previous point.
5. For all formulas built by using Boolean connectives or X, this is proven by induction on the structure of the formula and the closure of weak automata under all Boolean operations.
   The remaining cases are formulas of the form $\varphi_O \mathsf{U} \varphi_G$, $\varphi_O \mathsf{R} \varphi_S$, $\varphi_S \mathsf{W} \varphi_O$, and $\varphi_G \mathsf{M} \varphi_O$. We concentrate on formulas of the form $\varphi_O \mathsf{U} \varphi_G$. The others are similar. Consider a formula $\psi^o \mathsf{U} \psi^g$. We assume by induction that the automata obtained for ψ^o and ψ^g satisfy the Lemma. Consider the automaton created for $\psi^o \mathsf{U} \psi^g$. By structure of $\mathsf{tr}(\cdot)$, a run of D_ι has the following form:

$$
\begin{aligned}
&q_0 : \psi^o \mathsf{U} \psi^g\\
&q_1 : \left(\psi^o \mathsf{U} \psi^g \wedge \psi^o_{0,1}\right) \vee \psi^g_{0,1}\\
&q_2 : \left(\psi^o \mathsf{U} \psi^g \wedge \psi^o_{1,2} \wedge \psi^o_{0,2}\right) \vee \left(\psi^g_{1,2} \wedge \psi^o_{0,2}\right) \vee \psi^g_{0,2}\\
&q_3 : \left(\psi^o \mathsf{U} \psi^g \wedge \psi^o_{2,3} \wedge \psi^o_{1,3} \wedge \psi^o_{0,3}\right) \vee \left(\psi^g_{2,3} \wedge \psi^o_{1,3} \wedge \psi^o_{0,3}\right) \vee \left(\psi^g_{1,3} \wedge \psi^o_{0,3}\right) \vee \psi^g_{0,3}\\
&\dots\\
&q_i : \left(\psi^o \mathsf{U} \psi^g \wedge \bigwedge_{0\le j<i} \psi^o_{j,i}\right) \vee \bigvee\nolimits_{0\le j<i} \left(\psi^g_{j,i} \wedge \bigwedge_{1\le k<j} \psi^o_{k,i}\right),
\end{aligned}
$$

   where $\psi^\alpha_{j,k}$ represents the state of the automaton for ψ^α after having read the input $w_{j,k}$. Notice that formulas are simplified by the propositional equivalence. In particular, $\psi^\alpha_{j,k}$ could be equivalent to *tt* or *ff*, leading to a much simpler structure.
   Thus, the general form of a state of D_ι is (propositionally equivalent to):

$$\left(\psi^o \mathsf{U} \psi^g \wedge \bigwedge_{0\le j<i} \psi^o_{j,i}\right) \vee \bigvee_{0\le j<i} \left(\psi^g_{j,i} \wedge \bigwedge_{1\le k<j} \psi^o_{k,i}\right).$$

   We notice that for every state φ that has $\psi^o \mathsf{U} \psi^g$ as a component in the first disjunct, we have $\lambda(\varphi) \in \{\bot, \top\}$. Consider a state φ where the component containing $\psi^o \mathsf{U} \psi^g$ has been simplified to *ff* or *tt*. If state φ appears in a non-trivial SCC, then $\lambda(\varphi) \in \{\bot, \top\}$ as the same holds for the states of D_{ψ^o} and D_{ψ^g}.
   We have to show that every non-trivial SCC is assigned a uniform acceptance status of either $\bot$ or $\top$. Consider a non-trivial SCC where λ associates with some states $\bot$ and some states $\top$. Consider a loop that starts with λ being $\bot$, then $\top$, then $\bot$ again. The only states with which λ associates $\top$ are those for which $\lambda(\psi^g_{j,k}) = \top$. However, by induction, this is only the case if $\psi^g_{j,k} \equiv tt$. This is irreversible and hence the disjunct to which this $\psi^g_{j,k}$ belongs is a weak automaton that is obtained as the product of a fixed number of weak automata. By the smaller weak automata having every non-trivial SCC with uniform acceptance, it is impossible to have such an SCC with mixed acceptance. Hence, we conclude that the only way to go back

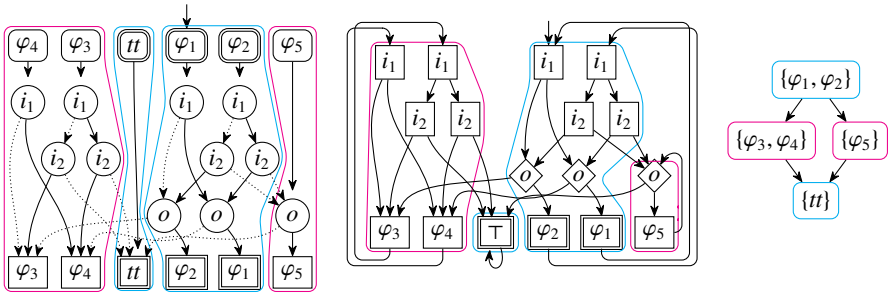

Fig. 5: The MTDWA from Figure 1, and its two-player game interpretation. Both also show their decomposition in accepting and rejecting SCCs. The rightmost picture is the "condensation graph" where each SCC has been reduced to a single node (labeled by the terminals it contains for easy identification).

from λ associating $\top$ to $\bot$ is if some conjunct containing $\psi^g_{j,k} \equiv tt$ simplifies overall to ff. That is, $\bigwedge_{1\leq k<j} \psi^o_{k,i} \equiv ff$. We notice that the first disjunct, which contains $\psi^o \cup \psi^g$, includes $\bigwedge_{1\leq k<j} \psi^o_{k,i}$ as a conjunct. Hence, the first disjunct, overall, becomes equivalent to ff as well. It follows that every non-trivial SCC with both accepting and rejecting states is part of the product of a fixed number of weak automata. By weak automata being closed under Boolean operations this is impossible.

□

E Synthesis Example

Figure 5 shows how an MTDWA can be interpreted as a two-player game with weak objective. Here we are doing synthesis with Mealy semantics, so the input variables are ordered above the output variables.

In the game interpretation, each internal node is assigned either to the input player (rectangular nodes), or to the output player (diamond nodes) based on the nature of the variable labeling that node. The terminal nodes can be assigned to either player since they do not allow any choice: from a terminal $\boxed{\alpha}$, the only possible move is to jump to the node corresponding to the root of $\Delta(\alpha)$.

The game is won by the output player iff it has a strategy to force the game to reach the accepting terminal (tt, φ_1, or φ_2) infinitely often. In terms of SCCs, the game is won by the output player iff it has a strategy to force the game to get stuck into an accepting SCC ($\{tt\}$, or $\{\varphi_1, \varphi_2\}$).

Before we solve the game, let us first address the fact that the condensation graph shown here does not contain an edge between φ_2 and tt, despite the fact that the explicit automaton from Figure 1 has one. If we are in state φ_2 and the input player plays $i_1 i_2$ or $\bar{i_1} i_2$, then from the point of view of the output player, the game can be played exactly as if we were in state φ_5. The MTDWA representation captures this, and the paths between φ_2 and $\top$ will temporarily enter the SCC of φ_5. This is inconsequential in practice since it will not affect the topological ordering of the SCCs.

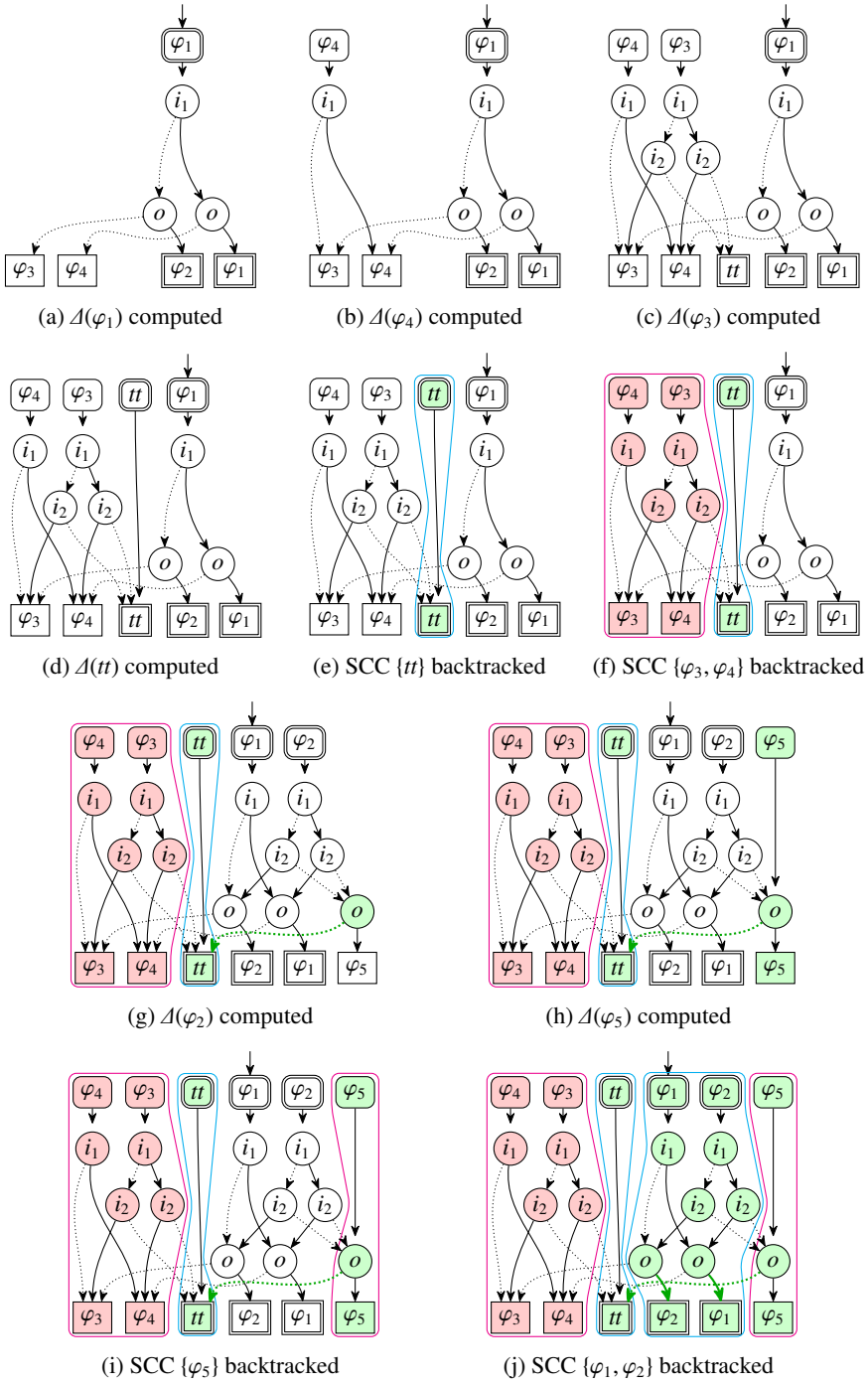

Fig. 6: Solving the example from Figure 1 as a game, while it is being constructed.

Figure 6 shows the step-by-step construction of our example MTDWA, solving it as a game along the way. The automaton is explored in DFS order because the goal is to find strongly-connected components.

In the first steps (a)–(d), the DFS explores states φ_1, φ_4, φ_3, and tt in that order. For each of those, $\Delta(\varphi_i)$ is computed (remember that $\Delta(\varphi_i)$ is $\mathsf{tr}(\varphi_i)$ from Section 3.2, followed by propositional equivalence from Section 3.2). During this exploration, the SCC algorithm updates its data structures (not shown nor discussed here) for the edges that are discovered. For instance during step (c) we learn that φ_3 and φ_4 belong to the same component, but we do not know if this component is maximal. We know that a component is maximal when the DFS backtracks from it. This happens for the first time at step (e). At this point we know that the component containing $\{tt\}$ is maximal, and because $\lambda(tt) = \top$ we can mark any undetermined state as winning (green) for the output player. Marking the $\boxed{tt}$ terminal as winning will immediately attempt to backpropagate to previous states. Here, nothing happens, because the predecessors of $\boxed{tt}$ are nodes played by the input player, who still has other options. (Our previous work [24] describes the data-structures we use to perform backpropagation in an on-the-fly context.) At step (f), all successors of φ_3 and φ_4 have been explored, so we know the SCC containing $\{\varphi_3, \varphi_4\}$ is maximal. Since $\lambda(\varphi_3) = \lambda(\varphi_4) = \bot$ any undetermined node in that SCC (in this case, all of them) can be marked (in red) as losing for the output player. The DFS continues to explore the successors of φ_2 and φ_5 on steps (g) and (h). Step (g) has one interesting bit: when the rightmost node $\bigcirc{o}$ (who is played by the output player) is connected to $\boxed{tt}$ (who is already known to be winning), then $\bigcirc{o}$ is automatically marked as winning too. Any such determination is automatically backpropagated, but at step (g) the predecessor of $\bigcirc{o}$ still has other choices so nothing happens. Step (h) computes $\Delta(\varphi_5)$ and connects φ_5 to $\bigcirc{o}$. Since this is the only possible choice, φ_5 can be marked as winning too. At step (i), SCC $\{\varphi_5\}$ is backtracked, since $\lambda(\varphi_5) = \bot$, all its undetermined nodes (there are none of them) can be marked as losing. Finally, at step (j) the SCC for $\{\varphi_1, \varphi_2\}$ is backtracked, and all its undetermined nodes (all of them) are marked as winning.

In practice the construction can stop as soon as the initial state is found winning (resp. losing): this shows that its formula is realizable (resp. unrealizable). In this example, doing the construction on-the-fly did not help because the initial state was only found winning after building the entire automaton.

If the goal is synthesis, not just realizability, the winning nodes labeled by output variables should also remember the choice that made them winning. The green arrows in Figure 6 highlight this. A winning strategy can be found by "killing the other choice" from each winning output node, i.e., redirecting it to a rejecting *ff* node, as shown in Figure 7. The resulting Mealy machine is then simplified using techniques described in previous work [64].

F Artifact

An artifact archived on Zenodo [22] contains:

- A copy of Spot 2.15.1 that implements the presented techniques.

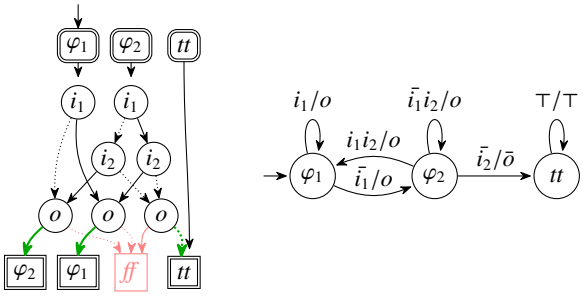

Fig. 7: Strategy extracted from Figure 6, represented using MTBDDs, or as an incompletely specified Mealy machine.

- A jupyter notebook showing how to reproduce Figures 1, 6, and 7.
- Source, results, and scripts to reproduce the two benchmarks presented in Section 5.
- Scripts to convert the benchmarks results into Figures 3, 4, 8, 9, and 11.
- A docker image allowing to execute all of the above, except the benchmark of Section 5.2 which uses BenchExec (cannot run in Docker).

G Detailed Translation Benchmark

This section digs a bit more into the translation benchmark summarized in Section 5.1. Recall that the formulas are of three different kinds. We use the following names:

scalable 310 formulas representing various instances of 17 scalable patterns [33,13,33,41,68]. These patterns are summarized in Table 2.

set 76 formulas collected from various works [25,67,28,57,36], these are the last six lines of Table 1.

syntcomp 108 unique formulas from the synthesis competition [37]: all instances that are syntactic obligations, except for two formulas syntactically equivalent to true or false. This is the first line of Table 1.

Figure 8 shows the runtime of four different configurations of `ltl2tgba` as well as that of `ltl2dela` and `ltl3tela`, over the different patterns of Table 2.

In these measurements, we can see that the new and new-min pipelines are almost indistinguishable, suggesting that the minimization is very cheap on these automata. We looked at the size of the automata (not shown here), to confirm that on these scalable patterns the new always produces a DWA of optimal size, making the minimization unnecessary in this case. The new and new-min pipelines also appear to be faster than everything else, often by some order of magnitude (the time scale is logarithmic!), with the exception of the `kr-n-delta1` and `kr-nlogn-delta1` formulas. On these two families, the old pipeline is faster. Profiling the code for `kr-nlogn-delta1` with $n = 3$ revealed that on these families the MTBDD-based translation applying propositional equivalence would build a 7559-state deterministic automaton that is then minimized to

Cf. App. H.

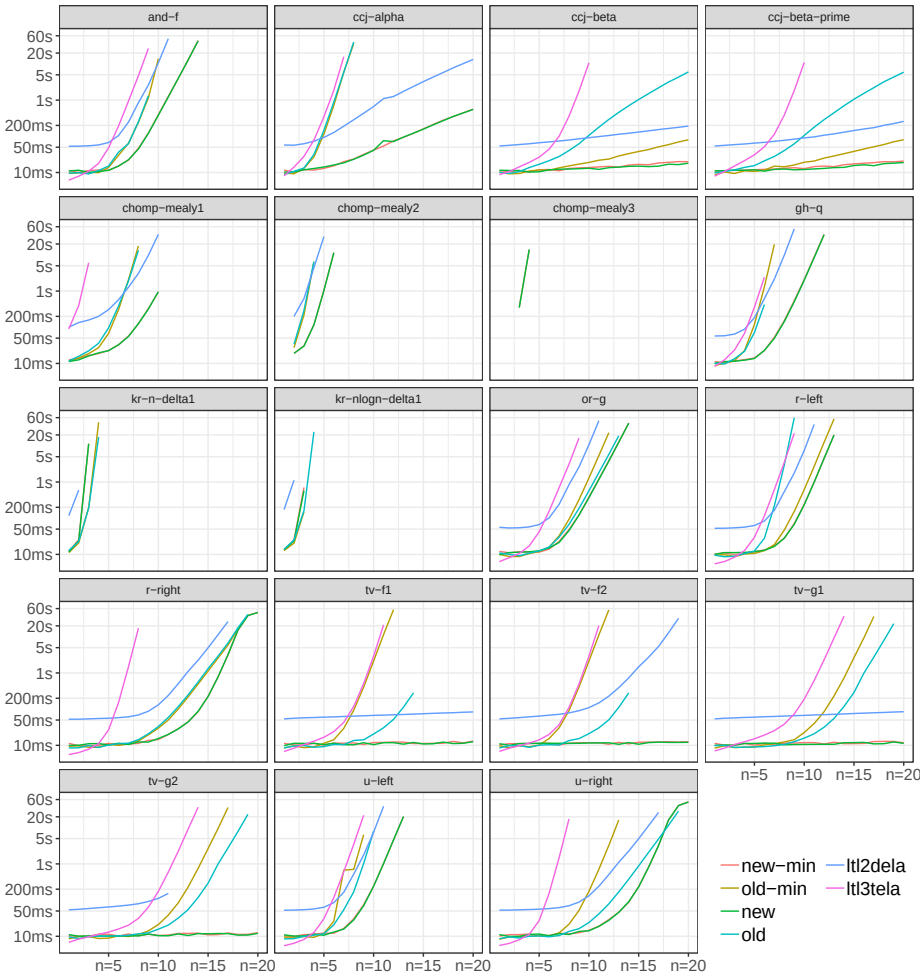

Fig. 8: Execution times of the different configurations, on the parametric formulas listed in Table 2

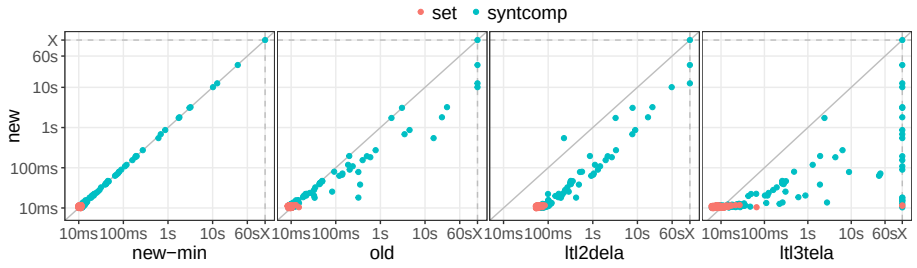

Fig. 9: Runtime comparisons over 184 non-scalable formulas from various sources.

Table 2: Parametric patterns used to evaluate the scalability of the translation. All these patterns can be generated by the `genltl` command of Spot. The class column shows the more precise class the pattern belongs to: (S) syntactic safety, (G) syntactic guarantee, or (O) syntactic obligation.

name	class	formula
`and-f` [33]	G	$\mathsf{F}(p_1) \wedge \mathsf{F}(p_2) \wedge \ldots \wedge \mathsf{F}(p_n)$
`ccj-alpha` [13]	G	$\mathsf{F}(p_1 \wedge \mathsf{F}(p_2 \wedge \mathsf{F}(p_3 \ldots \wedge \mathsf{F}(p_n)))) \wedge \mathsf{F}(q_1 \wedge \mathsf{F}(q_2 \wedge \mathsf{F}(q_3 \ldots \wedge \mathsf{F}(q_n))))$
`ccj-beta` [13]	G	$\mathsf{F}(p \wedge \mathsf{X}(p \wedge \mathsf{X}(p \ldots \wedge \mathsf{X}(p)))) \wedge \mathsf{F}(q \wedge \mathsf{X}(q \wedge \mathsf{X}(q \ldots \wedge \mathsf{X}(q))))$
`ccj-beta-prime` [13]	G	$\mathsf{F}(p \wedge \mathsf{X}(p) \wedge \mathsf{X}^2(p) \ldots \wedge \mathsf{X}^n(p)))) \wedge \mathsf{F}(q \wedge \mathsf{X}(q) \wedge \mathsf{X}^2(q) \ldots \wedge \mathsf{X}^n(q))$
`chomp-mealy`m	O	two-player $m \times n$ chomp game with Mealy semantics
`gh-q` [33]	O	$(\mathsf{F}(p_1) \vee \mathsf{G}(p_2)) \wedge (\mathsf{F}(p_2) \vee \mathsf{G}(p_3)) \wedge \ldots \wedge (\mathsf{F}(p_n) \vee \mathsf{G}(p_{n+1}))$
`kr-n-delta1` (H)	O	formula of size $\mathrm{O}(n)$, with doubly-exponential minimal DBA
`kr-nlogn-delta1`	O	formula of size $\mathrm{O}(n \log n)$, with doubly-exponential minimal DBA
`or-g` [33]	S	$\mathsf{G}(p_1) \vee \mathsf{G}(p_2) \vee \ldots \vee \mathsf{G}(p_n)$
`r-left`	S	$((p_1 \mathsf{R} p_2) \mathsf{R} p_3) \ldots \mathsf{R} p_n$
`r-right`	S	$p_1 \mathsf{R} (p_2 \mathsf{R} (\ldots \mathsf{R} p_n))$
`tv-f1` [68]	S	$\mathsf{G}(p \rightarrow (q \vee \mathsf{X}q \vee \mathsf{X}^2q \vee \ldots \vee \mathsf{X}^nq))$
`tv-f2` [68]	S	$\mathsf{G}(p \rightarrow (q \vee \mathsf{X}(q \vee \mathsf{X}(q \ldots \vee \mathsf{X}q))))$
`tv-g1` [68]	S	$\mathsf{G}(p \rightarrow (q \wedge \mathsf{X}q \wedge \mathsf{X}^2q \wedge \ldots \wedge \mathsf{X}^nq))$
`tv-g2` [68]	S	$\mathsf{G}(p \rightarrow (q \wedge \mathsf{X}(q \wedge \mathsf{X}(q \ldots \wedge \mathsf{X}q))))$
`u-left` [33]	G	$((p_1 \mathsf{U} p_2) \mathsf{U} p_3) \ldots \mathsf{U} p_n$
`u-right` [33]	G	$p_1 \mathsf{U} (p_2 \mathsf{U} (\ldots \mathsf{U} p_n))$

6207 states. Couvreur's algorithm also has a form of propositional equivalence, but it only builds a 1381-state NBA that is then determinized to 6454 states and minimized to 6207 states: a lot of time is saved because the propositional equivalence does not have to be performed during the determinization.

In families tv-f1, tv-f2, tv-g1, and tv-g2, the flat lines for new and new-min are mainly due to the fact that propositional equivalence is pushing X operators through Boolean formulas, as discussed in Section 3.2.

Figure 9 compares new against new-min, old, ltl2dela, and ltl3tela on 184 syntactic-obligation formulas from the set and syntcomp.

As can be seen on these plots, the set group of formulas is not very challenging.

For the syntcomp formulas, we can see again that the new translation is a clear improvement over old and the third-party tools ltl2dela and ltl3tela, and that minimization (new-min) incurs no overhead on these formulas.

It should also be pointed out that ltl3tela uses a portfolio approach that involves Spot already: it will first run its own algorithm, then run Spot's, and return the smallest result. Its runtime actually benefits from any improvement to the translation-pipeline of Spot in general. In the case of syntactic obligations, however, since the result of Spot is guaranteed to be minimal, ltl3tela's first attempt at translating the formula by other means is just unnecessary.

H `kr-n-delta1`

The following formula is a fixed version of a "worst-case LTL formula" for DBA translation, published by Kupferman & Rosenberg [41]. For a parameter n, the formula has size $O(n)$ but any equivalent DBA requires at least $O(2^{2^n})$ states.

$$\#\wedge \mathsf{X}\,(a_1 \vee b_1 \vee \$) \tag{1}$$

$$\wedge\, \mathsf{G}\left(\bigwedge_{i=1}^{n-1} ((a_i \vee b_i) \rightarrow \mathsf{X}\,(a_{i+1} \vee b_{i+1}))\right) \tag{2}$$

$$\wedge\, \mathsf{G}\,((a_n \vee b_n) \rightarrow \mathsf{X}\,(\# \wedge \mathsf{X}\,(a_1 \vee b_1 \vee \$ \vee \mathsf{G}\#))) \tag{3}$$

$$\wedge\, (\neg\$)\,\mathsf{U}\left(\$ \wedge \mathsf{X}\left((a_1 \vee b_1) \wedge \mathsf{X}^k\mathsf{G}\#\right)\right) \tag{4}$$

$$\wedge\, \mathsf{F}\left(\# \wedge \mathsf{X}\left(\neg\# \wedge \left(\bigvee_{i=1}^{n} ((a_i \wedge \mathsf{F}\,(\$ \wedge \mathsf{F}a_i)) \vee (b_i \wedge \mathsf{F}\,(\$ \wedge \mathsf{F}b_i)))\right)\mathsf{U}\,\#\right)\right) \tag{5}$$

$$\wedge\, \mathsf{G}\left((\# \vee \$) \rightarrow \neg \bigvee_{i=1}^{n} (a_i \vee b_i)\right) \wedge \mathsf{G}(\# \rightarrow \neg\$) \wedge \mathsf{G}\left(\bigwedge_{i=1}^{n} (a_i \rightarrow \neg b_i)\right) \tag{6}$$

This formula is not a syntactic obligation because equation (4) contains a G operator in the right operand of a U.

However, because proposition # will eventually occur continuously, equation (4) can be replaced by the following equivalent syntactic obligation:

$$\wedge\left((\neg\$)\,\mathsf{W}\left(\$ \wedge \mathsf{X}\left((a_1 \vee b_1) \wedge \mathsf{X}^k\mathsf{G}\#\right)\right)\right) \wedge \mathsf{F}(\#)$$

This formula also assumes a letter-alphabet. To use it in our context where the alphabet is propositional, we interpret each letter as a proposition, and add some constraints to ensure that only one proposition is true at any time, as discussed by Kupferman & Rosenberg [41].

This modified version is what we call `kr-n-delta1` in Table 2. `kr-nlogn-delta1` can be obtained by a similar modification to the quasilinear formula from the same paper.

I `ltlsynt`'s Architecture

Figure 10 shows the current architecture of `ltlsynt`. The green path at the top of the figure corresponds to the description of Section 4. It is what is described as (new synt.) in Figures 4 and 11.

The large blue area contains multiple ways to convert a specification into a parity game. The path highlighted in red is the current default of `ltlsynt`. It will be used if the specialized techniques do not apply, or if they are disabled (for instance by setting `--obligation-synthesis=no`).

The dashed nodes are all using the `spot::translator` class to create a deterministic automaton. They correspond to the dashed block in Figure 2, so they still benefit

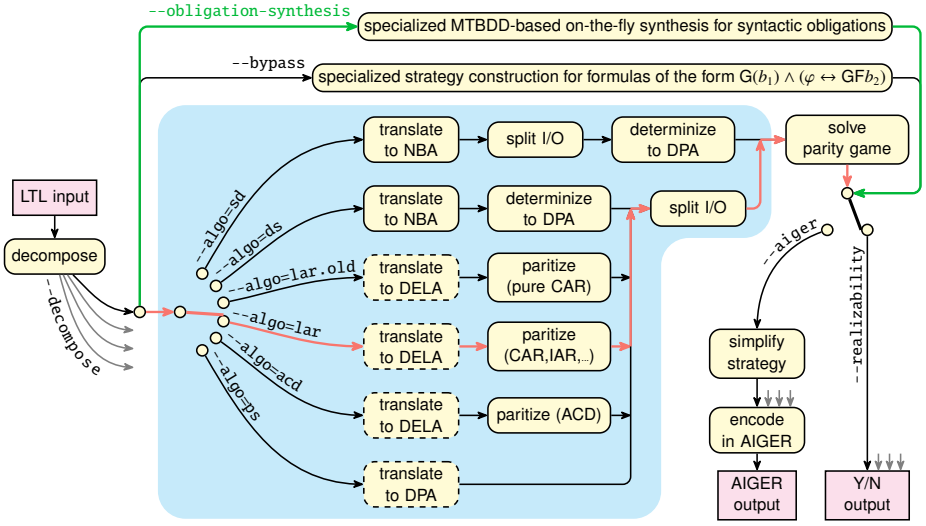

Fig. 10: Overview of `ltlsynt`'s architecture, taken from its documentation.

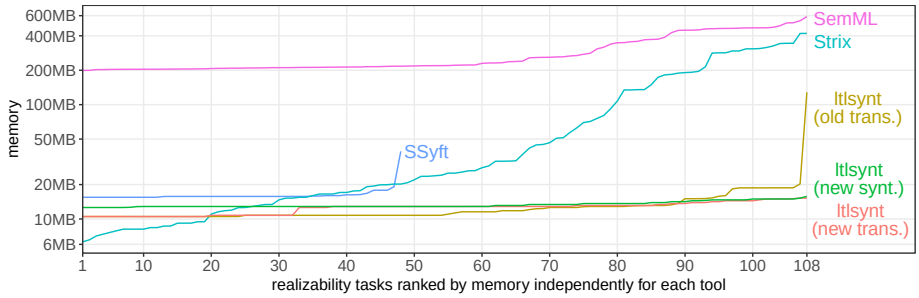

Fig. 11: Memory usage for the experiment of Figure 4.

from the translation improvement described in Section 3. In Figures 4 and 11, the (old trans.) and (new trans.) configurations correspond to taking the red path of Figure 10, and taking the old-min or new-min paths in the dashed block of Figure 2.

J Memory Usage

Since we used BenchExec for the experiments of Section 5.2, we can also report about the memory usage of each tested configuration. This is plotted in Figure 11.

K SemML

Our benchmark is not very favorable to SemML, because this tool is geared toward difficult specifications that are not solved in a few seconds, using Machine Learning techniques to guide the on-the-fly construction of the game.

The large runtime observed in Figure 4 can be attributed to the cost of starting a JVM, loading the ML models from files, and initiating the whole process from a Python script. For all other tools, a native binary is called.

SemML was the winner of the LTL realizability track of SyntComp'24. It was outranked by `ltlsynt` in SyntComp'25 only because of a small technical issue: its starting script passes the LTL specification on the command line. This fails for specifications that are larger than what a command line can accommodate. SemML failed to process at least 50 specifications because of this issue. `ltlsynt` had no problem reading these large formulas, because it read them from a pipe.

Those large specifications are not syntactic obligations, so this was not a problem for our benchmark.

L Obligations After Decomposition

We mention in Section 6 that some synthesis specifications can be decomposed into a conjunction of output-disjoint sub-specifications that can be solved independently [30,65]. If the original specification is not a syntactic obligation, it was not included in our benchmark. However, it could be the case that some of the sub-specifications are syntactic obligations. Although we did not evaluate those, they will benefit from the improved game solving. Table 3 shows the list of such specifications in the SyntComp benchmark collection. For instance `Gamelogic.tlsf` contains 4 sub-specifications, and 3 of those are syntactic obligations.

file	obligations/subs
tsl_paper/SensorInit.tlsf	1/2
tsl_paper/Gamelogic.tlsf	3/4
tsl_paper/KitchenTimerV7.tlsf	1/2
tsl_paper/ModdifiedLedMatrix4X.tlsf	2/3
tsl_paper/KitchenTimerV8.tlsf	2/3
tsl_paper/KitchenTimerV9.tlsf	1/2
tsl_paper/ModdifiedLedMatrix5X.tlsf	2/3
tsl_paper/KitchenTimerV10.tlsf	1/2
tsl_smart_home_jarvis/.../Light_a50cadd7.tlsf	1/2
tsl_smart_home_jarvis/.../test4_4b82b1f4.tlsf	2/3
tsl_smart_home_jarvis/.../Room_a50cadd7.tlsf	1/3
tsl_smart_home_jarvis/.../test4_4b82b1f4_1.tlsf	1/2
tsl_smart_home_jarvis/.../Example1_d6376bf9.tlsf	2/3
tsl_smart_home_jarvis/.../jarvis_philippe_484face8_2.tlsf	4/5
tsl_smart_home_jarvis/.../Example.tlsf	1/2

Table 3: List of SyntComp specifications that are not syntactic obligations, but that can be decomposed into output-disjoint sub-specifications in which there are syntactic obligations.