Fast Obligation Translation and Synthesis

Citation for the original published paper (version of record):
Duret-Lutz, A., De Giacomo, G., Jurdzinski, M. et al (2026). Fast Obligation Translation and Synthesis. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 16682 LNCS: 322-339.
http://dx.doi.org/10.1007/978-3-032-32519-8_17

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
adl@lre.epita.fr
[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, TX, 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) [60], and converts them to strategies that ensure their satisfaction [61]. 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 [30,43]. Two major successes in this field have been the syntactic fragment of LTL called GR[1] [59], and $\mathrm{LTL_f}$ [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 $\mathrm{LTL_f}$ synthesis, tools leverage the algorithmic simplicity of deterministic finite automata (DFA), in contrast to ω-automata [4,16,45,70].

Obligation properties [51], 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, obli-

E. Darulova et al. (Eds.): CAV 2026, LNCS 16682, pp. 322–339, 2026.
https://doi.org/10.1007/978-3-032-32519-8_17

gation 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. [26] 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 [38] at the 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 [51–54]. Recently, the safety-progress hierarchy has attracted renewed interest thanks to a normalization procedure that reduces the nesting depth of strong and weak operators [28].

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) [46,47], 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 [47], they can also be minimized like DFAs.

Unfortunately, testing whether an LTL formula is an obligation property is 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* [10,11]. 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 [25]. 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 [24]. 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.

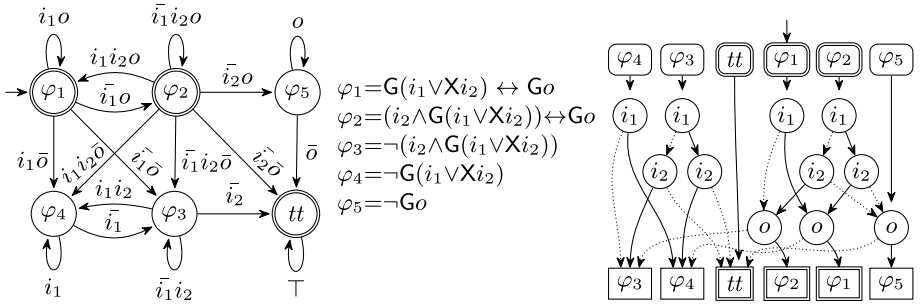

Fig. 1. A DBA for $\varphi_1 = \mathsf{G}(i_1 \vee \mathsf{X}i_2) \leftrightarrow \mathsf{G}o$, and its MTBDD representation.

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 Fig. 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 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 Fig. 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 [36,39], and previous work [25]. The reader familiar with binary decision diagrams (BDDs) [8] will recognize the classical structure of a forest of BDDs: $(x) \overset{\ell}{\underset{h}{\rightrightarrows}}$ 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 [32,39,48,55] 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* $\mathcal{A} = \langle \mathcal{Q}, \mathcal{P}, \iota, \Delta, \lambda \rangle$*, where* $\mathcal{Q}$ *is a finite set of* states*,* $\mathcal{P}$ *is a finite (and ordered) set of* variables*,* $\iota \in \mathcal{Q}$ *is the initial state,* $\Delta : \mathcal{Q} \to \mathsf{MTBDD}(\mathcal{P}, \mathcal{Q})$ *represents the outgoing edges of each state, and* $\lambda : \mathcal{Q} \to \mathbb{B}$ *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 Fig. 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 [60]. The following grammar (where parentheses have been omitted, and $p \in \mathcal{P}$) defines four syntactic fragments [10,11]:

$$\varphi_B ::= ff \mid tt \mid p \mid \neg\varphi_B \mid \varphi_B \wedge \varphi_B \mid \varphi_B \vee \varphi_B \mid \varphi_B \leftrightarrow \varphi_B \mid \varphi_B \oplus \varphi_B \mid \varphi_B \rightarrow \varphi_B \mid \mathsf{X}\varphi_B$$
$$\varphi_G ::= \varphi_B \mid \neg\varphi_S \mid \varphi_G \wedge \varphi_G \mid \varphi_G \vee \varphi_G \mid \varphi_S \rightarrow \varphi_G \mid \mathsf{X}\varphi_G \mid \mathsf{F}\varphi_G \mid \varphi_G \,\mathsf{U}\, \varphi_G \mid \varphi_G \,\mathsf{M}\, \varphi_G$$
$$\varphi_S ::= \varphi_B \mid \neg\varphi_G \mid \varphi_S \wedge \varphi_S \mid \varphi_S \vee \varphi_S \mid \varphi_G \rightarrow \varphi_S \mid \mathsf{X}\varphi_S \mid \mathsf{G}\varphi_S \mid \varphi_S \,\mathsf{R}\, \varphi_S \mid \varphi_S \,\mathsf{W}\, \varphi_S$$
$$\varphi_O ::= \varphi_G \mid \varphi_S \mid \neg\varphi_O \mid \varphi_O \wedge \varphi_O \mid \varphi_O \vee \varphi_O \mid \varphi_O \leftrightarrow \varphi_O \mid \varphi_O \oplus \varphi_O \mid \varphi_O \rightarrow \varphi_O \mid \mathsf{X}\varphi_O$$
$$\mid \varphi_O \,\mathsf{U}\, \varphi_G \mid \varphi_O \,\mathsf{R}\, \varphi_S \mid \varphi_S \,\mathsf{W}\, \varphi_O \mid \varphi_G \,\mathsf{M}\, \varphi_O$$

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. [28]. 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 [10,11].

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 $\langle \mathcal{Q}, \mathcal{P}, \iota, \Delta, \lambda \rangle$, where states $\mathcal{Q} \subseteq \mathsf{LTL}_O(\mathcal{P})$ are identified by syntactic obligations, and $\iota \in \mathsf{LTL}_O(\mathcal{P})$ is the formula 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:

$$\mathsf{tr}(tt) = \boxed{tt} \qquad \mathsf{tr}(ff) = \boxed{ff} \qquad \mathsf{tr}(a) = \begin{array}{c} (a) \\ {\scriptstyle 0}\swarrow \quad \searrow{\scriptstyle 1} \\ \boxed{ff} \quad \boxed{tt} \end{array} \text{ for any } a \in \mathcal{P}$$

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

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

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

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

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

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.

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

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

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

The special value $*$ should be ignored during Boolean operations, as shown below:

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

A state φ such that $\lambda(\varphi) = *$ is a Boolean combination of atomic propositions, possibly with X operators. It cannot be part of a cycle, so 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 $\langle \mathcal{Q}, \mathcal{P}, \iota, \Delta, \lambda \rangle$ by letting $\Delta = \mathsf{tr}$, and by setting $\mathcal{Q}$ to be equal to all the formulas that can be reached transitively from ι by applying tr. This does not work because the set $\mathcal{Q}$ constructed this way is not necessarily finite. The classical solution, which we reuse, is to use propositional equivalence [25,27].

In previous work on $\mathrm{LTL_f}$ translation [25], 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 $\mathcal{Q}$, 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.

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

1. *Any state $\varphi \in \mathcal{Q}$ such that $\lambda(\varphi) = *$ 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 $\varphi \in \mathcal{Q} \setminus \{tt\}$ 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 $\varphi \in \mathcal{Q} \setminus \{ff\}$ 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 $\bot$.*

Proofs and supplementary material can be found in the extended version of this article, available on arXiv [23].

Since the states φ for which $\lambda(\varphi) = *$ cannot be part of a cycle, their acceptance can be chosen arbitrarily. In Sect. 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 $\lambda' : \mathcal{Q} \to \mathbb{B}$ 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 $D_\iota = \langle \mathcal{Q}, \mathcal{P}, \iota, \mathsf{tr}, \lambda' \rangle$ constructed above is an MTDWA, and it satisfies $\mathscr{L}(\iota) = \mathscr{L}(D_\iota)$.*

Proof. The fact that D_ι is weak is a corollary that follows directly from Lemma 1. $\mathscr{L}(D_\iota)$ being equivalent to $\mathscr{L}(\iota)$ 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 [47]. 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 [47] 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 [56] similar to what is implemented in Mona [36]. 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 ([30,43]). *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 $(\sigma \sqcup \sigma_\rho) \in \mathscr{L}(\varphi)$ using the desired semantics.*

Formula φ_1 (from Fig. 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 [44, 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.

We implement this game solving over MTDWA on-the-fly, together with the automaton construction of Sect. 3.

The setup is very similar to our previous solution for $\mathrm{LTL_f}$ synthesis [25] 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 [33,35,63]. 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 [25, Algorithm 1].

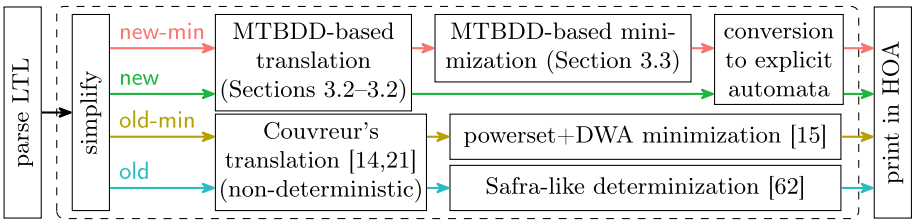

Fig. 2. Four pipelines usable by `ltl2tgba` to translate a syntactic obligation: new-min, new, old-min, and old. The `spot::translator` class implements the dashed box.

5 Implementation and Evaluation

The new constructions for translation, minimization, and game solving have been implemented [22] in Spot [24] 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 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 [62].

In both old-min and old, automata are represented as explicit graphs with edges labeled by Boolean formulas over propositions (represented as BDDs) [21, 24], as on the left-hand side of Fig. 1. In the new pipelines, called new-min and new, the MTBDD-based translation described in Sect. 3 produces an MTDWA as on the right-hand side of Fig. 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 [50] and ltl2dela from Owl 21.0 [41]. (The latter is an evolution of Delag [57], 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 s. 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 1.6 GHz to prevent CPU throttling.

We evaluated the translation pipelines on 494 syntactic obligations:

- 310 formulas representing instances of 17 scalable patterns [13,34,42,68];
- 76 formulas collected from various works [26,29,37,58,67];
- 108 unique formulas from the synthesis competition [38]: 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.

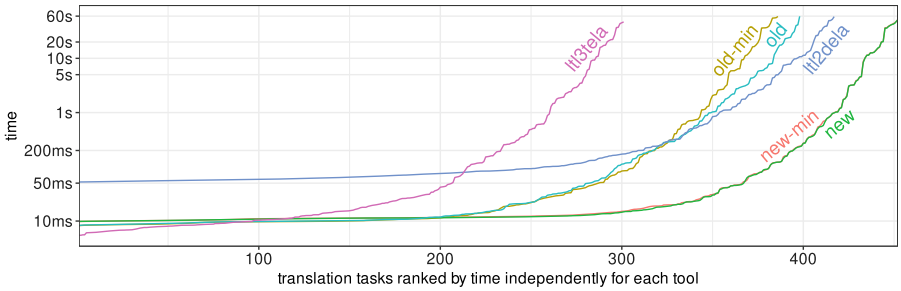

Fig. 3. Comparison of translation tools on 494 syntactic obligations from the literature.

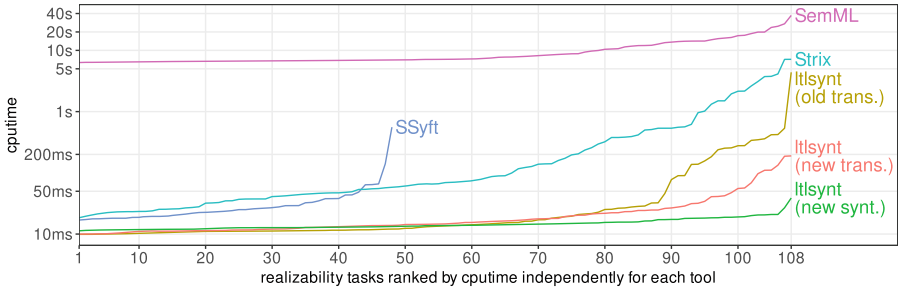

Fig. 4. Comparison of the three configurations of `ltlsynt` over the 108 specifications from SyntComp that are syntactic obligations. Third-party tools Strix, SemML, and SSyft are included for reference.

5.2 Synthesis

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 Fig. 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 Fig. 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.

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 [49] (the winner of SyntComp'23) and SemML [40] (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.

Additional details and discussion can be found in the extended version of this article, available on arXiv [23].

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 [31,65]. 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 [9,64]. To obtain a DELA, Spot generalizes the compositional translation used by the `Delag` tool [57]: 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.

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 [28], converting top-level temporal formulas to deterministic Büchi and co-Büchi automata, and using Boolean combinations for the final automaton.

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.

Data Availibility Statement. The source code for Spot 2.15.1, its documentation, benchmark data and scripts, some illustrative notebooks, and a Docker image containing an installation of Spot and all third-party tools measured in our benchmark, are archived on Zenodo [22].

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: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 870–893. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_41
2. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (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 2022), pp. 1–19. Springer (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 2020), 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. Int. J. Softw. Tools Technol. Transfer **21**, 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
6. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report 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. J. Comput. Syst. Sci. **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 Trans. Comput. **35**(8), 677–691 (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: TACAS 2022. LNCS, vol. 13244, pp. 99–117. Springer, Cham (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.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45138-9_26
11. Chang, E., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 474–486. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55719-9_97
12. Chatterjee, K.: Linear time algorithm for weak parity games. Technical Report UCB/EECS-2006-153, Electrical Engineering and Computer Sciences, University of California at Berkeley (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 2009), 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 linear temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (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.) ATVA 2007. LNCS, vol. 4762, pp. 223–236. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75596-8_17
16. De Giacomo, G., Favorito, M.: Compositional approach to translate $\mathrm{LTL}_f/\mathrm{LDL}_f$ into deterministic finite automata. In: Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021), 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 2013), pp. 854–860. AAAI Press (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 2015), 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 (1973). http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF
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. Int. J. Crit. Comput.-Based Syst. **5**(1/2), 31–54 (2014). https://doi.org/10.1504/IJCCBS.2014.059594
22. Duret-Lutz, A.: Supporting material for "Fast Obligation Translation and Synthesis" (2026). https://doi.org/10.5281/zenodo.19812382
23. Duret-Lutz, A., De Giacomo, G., Jurdzinski, M., Piterman, N., Vardi, M.Y., Zhu, S.: Fast obligation translation and synthesis. arXiv (2026). https://doi.org/10.48550/arXiv.2605.12372, extended version of this article, inluding proofs and additional discussions in appendices
24. Duret-Lutz, A., et al.: 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, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
25. 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 2025). Lecture Notes in Computer Science, vol. 15981, pp. 129–147. Springer (2025). https://doi.org/10.1007/978-3-032-02602-6_10
26. 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 1998), pp. 7–15. ACM Press (1998). https://doi.org/10.1145/298595.298598
27. 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 2018), pp. 384–393. ACM (2018). https://doi.org/10.1145/3209108.3209161
28. Esparza, J., Rubio, R., Sickert, S.: Efficient normalization of linear temporal logic. J. ACM **71**(2) (2024). https://doi.org/10.1145/3651152

29. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–168. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_13
30. 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
31. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 113–130. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_8
32. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. Formal Methods Syst. Des. **10**(2/3), 149–169 (1997). https://doi.org/10.1023/A:1008647823331
33. Gaiser, A., Schwoon, S.: Comparison of algorithms for checking emptiness on Büchi automata. In: Hlinený, P., Matyás, V., Vojnar, T. (eds.) Proceedings of Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09). OASICS, vol. 13. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany (2009). https://doi.org/10.4230/DROPS.MEMICS.2009.2349
34. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 53–70. Springer, Heidelberg (2006). https://doi.org/10.1007/11691617_4
35. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. **345**(1), 60–82 (2005). https://doi.org/10.1016/j.tcs.2005.07.004
36. Henriksen, J.G., et al.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5
37. Holeček, J., Kratochvíla, T., Řehák, V., Šafránek, D., Šimeček, P.: Verification results in Liberouter project. Technical Report 03, CESNET (2004). http://archiv.cesnet.cz/doc/techzpravy/2004/verificationresults/
38. Jacobs, S., et al.: The reactive synthesis competition (SYNTCOMP): 2018–2021. arXiV (2022). https://doi.org/10.48550/ARXIV.2206.00251
39. Klarlund, N., Møller, A.: MONA version 1.4, user manual. Technical report, BRICS (2001). https://www.brics.dk/mona/mona14.pdf
40. 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 2025), pp. 233–253. Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_12
41. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
42. Kupferman, O., Rosenberg, A.: The blow-up in translating LTL to deterministic automata. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS (LNAI), vol. 6572, pp. 85–94. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20674-0_6

43. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), pp. 531–542 (2005). https://doi.org/10.1109/SFCS.2005.66
44. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000). https://doi.org/10.1145/333979.333987
45. 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 2025), pp. 1711–1718. Frontiers in Artificial Intelligence and Applications (2025). https://doi.org/10.3233/FAIA250999
46. 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
47. Löding, C.: Efficient minimization of deterministic weak ω-automata. Inf. Process. Lett. **79**(3), 105–109 (2001). https://doi.org/10.1016/S0020-0190(00)00183-6
48. Long, D.: BDD library. https://www.cs.cmu.edu/~modelcheck/bdd.html
49. 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
50. Major, J., Blahoudek, F., Strejček, J., Sasaráková, M., Zbončáková, T.: `ltl3tela`: LTL to small deterministic or nondeterministic Emerson-lei automata. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 357–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_21
51. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC 1990), pp. 377–410. ACM, New York, NY, USA (1990). https://doi.org/10.1145/93385.93442
52. 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
53. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer (1995). https://doi.org/10.1007/978-1-4612-4222-2
54. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: response. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 279–361. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13754-9_13
55. Minato, S.I.: Representation of Multi-Valued Functions, pp. 39–47. Springer, Boston (1996). https://doi.org/10.1007/978-1-4613-1303-8_4
56. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies. Annals of Mathematical Studies, no. 34, pp. 129–153. Princeton University Press (1956)
57. Müller, D., Sickert, S.: LTL to deterministic Emerson-Lei automata. In: Proceedings of the 8th International Symposium 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
58. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_17
59. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_24

60. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
61. 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 1989). Association for Computing Machinery (1989). https://doi.org/10.1145/75277.75293
62. Redziejowski, R.: An improved construction of deterministic omega-automaton using derivatives. Fund. Inform. **119**(3–4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744
63. 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.) LPAR 2013. LNCS, vol. 8312, pp. 668–682. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_44
64. Renkin, F., Duret-Lutz, A., Pommellet, A.: Practical "paritizing" of Emerson-lei automata. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 127–143. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_7
65. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting `ltlsynt`. Formal Methods Syst. Des. (2023). https://doi.org/10.1007/s10703-022-00407-6
66. Schewe, S.: Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete. In: Lodaya, K., Mahajan, M. (eds.) Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). Leibniz International Proceedings in Informatics, vol. 8, pp. 400–411. Schloss Dagstuhl LZI, Dagstuhl, Germany (2010). https://doi.org/10.4230/LIPIcs.FSTTCS.2010.400
67. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_21
68. Tabakov, D., Vardi, M.Y.: Optimized temporal monitors for SystemC. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 436–451. Springer, Heidelberg (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 1985), 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 2017), pp. 1362–1369 (2017). https://doi.org/10.24963/ijcai.2017/189