Compositional Approach to Translate $\text{LTL}_f/\text{LDL}_f$ into Deterministic Finite Automata

Giuseppe De Giacomo and Marco Favorito
DIAG - University of Rome “La Sapienza”, Italy
{degiacomo.favorito}@diag.uniroma1.it

Abstract
The translation from temporal logics to automata is the workhorse algorithm of several techniques in computer science and AI, such as reactive synthesis, reasoning about actions, FOND planning with temporal specifications, and reinforcement learning with non-Markovian rewards, just to name a few. Unfortunately, the problem is computationally intractable, requiring the implementation of several heuristics to make it usable in practice. In this paper, following the recent interest in temporal logic formalisms over finite traces, we present a compositional approach for dealing with translations of Linear Temporal Logic and Linear Dynamic Logic ($\text{LDL}_f$) on finite traces into Deterministic Finite Automata (DFA). That is, we inductively transform each $\text{LTL}_f/\text{LDL}_f$ sub-formula into a DFA, and combine them through automata operators. By relying on efficient semi-symbolic automata representations, we empirically show the effectiveness of our approach and the competitiveness with similar tools. Moreover, this is the first work that provides a scalable and practical tool supporting the translation to DFA not only for $\text{LTL}_f$ but also for full $\text{LDL}_f$.

Introduction
Linear Temporal Logics over finite traces ($\text{LTL}_f$), and its extension with regular expression, Linear Dynamic Logic ($\text{LDL}_f$) (De Giacomo and Vardi 2013), are important logic formalisms extensively used in Artificial Intelligence and Computer Science. For example, it is used in reactive synthesis (De Giacomo and Vardi 2015, 2016; Camacho et al. 2018; Zhu et al. 2017), in FOND planning with temporal specifications (Brafman and De Giacomo 2019a; Camacho and McIlraith 2019), to express trajectory constraints in PDDL 3.0 (Bacchus and Kabanza 1998; Gerevini et al. 2009), in the theory of Markov Decision Processes to capture non-Markovian rewards (Bacchus, Boutilier, and Grove 1996; Brafman, De Giacomo, and Patrizi 2018; Brafman and De Giacomo 2019b) with applications in reinforcement learning (Camacho et al. 2019; De Giacomo et al. 2019, 2020a), to specify business processes (Pešić, Bošnački, and van der Aalst 2010), and many others.

Reasoning over $\text{LTL}_f/\text{LDL}_f$ is usually done by relying on automata theory. In particular, from a $\text{LTL}_f/\text{LDL}_f$ formula $\varphi$, we can build a deterministic finite automaton (DFA) $A_{\varphi}$, whose alphabet is the set of propositional interpretations $P$ of $\varphi$, that is semantically equivalent to the original formula (De Giacomo and Vardi 2013, 2015). The computational complexity of such translation has been shown to be doubly exponential time in the worst case, and indeed $A_{\varphi}$ can be double-exponentially larger than the original formula $\varphi$. Nevertheless, in most cases the resulting DFA is actually manageable, a phenomenon often observed when determinization is applied to automata finite words. (Tabakov and Vardi 2005). This puts working in the finite traces in sharp contrast with working with infinite ones, which are hampered by the notorious intractability of determinization of nondeterministic Büchi automata (Fogarty et al. 2015).

One of the ingredients of the translation from such logics to DFAs is the Mona tool (Henriksen et al. 1995; Klrlund 1997; Klarlund, Møller, and Schwartzbach 2001). The tool implements the translation from First-Order Logic (FO) and Monadic Second-Order Logic on finite strings (MSO) to deterministic finite automata. Thanks to its novel and efficient semi-symbolic representation, still explicit in the state space’s representation but symbolic in the transitions, Mona has become widely used in the research community. One of the best practical implementation of the translation from $\text{LTL}_f$ to DFA, proposed by (Zhu et al. 2017). Their tool Syft encodes $\text{LTL}_f$ formulae into First-Order Logic formulae, represented as Mona programs, and uses Mona to perform the actual translation. The Mona output is then post-processed to produce a fully symbolic representation (i.e. both in the state space and in the transitions) to perform $\text{LTL}_f$ synthesis. A more recent work (Bansal et al. 2020) proposed a hybrid approach to the problem of DFA construction from $\text{LTL}_f$ formulae: first, they decompose the outermost conjunction in $\varphi$, where $\varphi$ is assumed to be in the form $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, in n-subformulae $\varphi_1, \ldots, \varphi_n$. Then, they transform each $\varphi_i$ into DFAs $A_{\varphi_i}$, in explicit-state representation using Mona. Finally, they start doing the product between all the automata $A_{\varphi_i}$; if at some point the size of the partial automaton becomes too large and exceeds a user-defined threshold, the approach converts all the explicit-state automata in symbolic representation and continues with the products, though forgoing minimization. In this way the tool is able to scale even in the case the automaton becomes prohibitively large to be represented explicitly, although not producing a minimal automaton anymore in this case. Both
tools in (Zhu et al. 2017) and in (Bansal et al. 2020) perform much better than state of the art tools, such as SPOT (Duret-Lutz et al. 2016), which implement procedures to translate LTL formulae to automata on infinite words, and can also be used for LTL by exploiting its encoding into LTL (De Giacomo and Vardi 2013). They implemented a tool called Lisa and LisaSynt, for DFA translation and synthesis, respectively.

Observe that both tools make use of the translation of FO into DFA, provided by Mona, which is nonelementary \(^1\) in the worst case, due to the necessity of multiple determinizations (each exponential in the worst case) and projections (which introduces nondeterminism) needed to handle quantifiers and negations. Still, this non-elementariness does not show in practice (again for the phenomenon of determinization of automata on finite words mentioned above).

In this work, we take a step further from the compositional approach proposed in (Bansal et al. 2020). In particular, our contribution is a fully compositional approach to handle both LTL\(_f\) formulae and LDL\(_f\) formulae. That is, we don’t make any assumption on the structure of the formula, as done by Bansal et al. which stops the decomposition step at the outermost conjunction. We process all the subformulae recursively up to the leaves of the syntax tree, and then we compose the partial DFAs of the subformulae using common operations over automata (e.g. union, intersection, concatenation), according to the LTL\(_f\)/LDL\(_f\) operator being processed.

Our contribution is both theoretical and practical. On the theoretical side, we observe that so far the theory of the correspondence between LTL\(_f\)/LDL\(_f\) and automata theory relied on the transformation of LTL\(_f\)/LDL\(_f\) formulae into Alternating Automata on finite words (AFA), which can be eventually transformed into Nondeterministic Finite Automata (NFA), and in turn determinized into DFAs (De Giacomo and Vardi 2013). Instead, we provide a sound and complete technique to directly transform a formula into a DFA. Despite the worst-case complexity of such technique is again nonelementary, as Mona’s, we show that it has several practical advantages with respect to the previous ones, primarily due to the possibility to apply aggressive minimization to the partial automata, which has already been argued to be indispensable for scalability (Klarlund, Møller, and Schwartzbach 2001; Zhu et al. 2020). On the practical side, we provide an implementation that employs such a compositional technique, and showing its competitiveness with existing tools (Bansal et al. 2020; Henriksen et al. 1995). Our tool can be used both for LTL\(_f\)/LDL\(_f\)-to-DFA construction, and as a LTL\(_f\)/LDL\(_f\) synthesis tool. Crucially, this is the first work that provides a scalable and practical tool supporting the translation to DFA and synthesis not only for LTL\(_f\) but also for full LDL\(_f\).

---

\(^1\)In computational complexity theory, a nonelementary problem is a problem that is not a member of the ELEMENTARY class. In other words, the computational time cost of such problems has an unbounded number of exponentiations.

---

### Preliminaries

**LTL\(_f\)** and **LDL\(_f\)**. LTL\(_f\) and LDL\(_f\) are, respectively, Linear Temporal Logic and Linear Dynamic Logic with finite trace semantics, proposed in (De Giacomo and Vardi 2013). LTL\(_f\) shares the same syntax of LTL (Pnueli 1977). It is as expressive as First-Order Logic over finite traces, so strictly less expressive than regular expressions, which, in turn, are as expressive as Monadic Second-Order logic over finite traces.

The semantics of LTL\(_f\) (and LDL\(_f\)) is given in terms of finite traces denoting a finite, possibly empty, sequence \(\pi = \pi_0, \ldots, \pi_n\) of elements from the alphabet \(2^P\), containing all possible propositional interpretations of the propositional symbols in \(P\). We denote the length of the trace \(\pi\) as \(\text{length}(\pi) = n + 1\), and with \(\text{last}(\pi) = n\) the last index. We denote as \(\pi(i) = \pi_i\) the \(i\)-th step in the trace. If the trace is shorter and does not include an \(i\)-th step, \(\pi(i)\) is undefined. We denote by \(\pi(i)\) the \(i\)-th segment of the trace \(\pi\) starting at the \(i\)-th step and ending at the \(j\)-th step (excluded). If \(j > \text{length}(\pi)\) then \(\pi(i, j) = \pi(i, \text{length}(\pi))\). For every \(j \leq i\), we have \(\pi(i, j) = \epsilon\), i.e., the empty trace. Notice that, differently from (De Giacomo and Vardi 2013), we allow the empty trace as in (Brafman, De Giacomo, and Patrizi 2018).

Given a set \(P\) of propositional symbols, LTL\(_f\) formulae are built as follows:

\[
\varphi ::= \phi \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 U \varphi_2
\]

where \(\phi\) is a propositional formula over \(P\), \(\bigcirc\) is the next operator, and \(U\) is the until operator. In addition, we have common abbreviations. For example, \(\bigcirc\) is the weak next operator, for which we have the equivalence \(\bullet \varphi \equiv \neg \bigcirc \neg \varphi\) (notice that in the finite trace case \(\neg \bigcirc \neg \varphi = \bigcirc \neg \varphi\), \(R\) is release operator, for which we have the equivalence \(\varphi_1 R \varphi_2 \equiv \neg (\neg \varphi_1 U \neg \varphi_2)\). Eventually \(\bigcirc \varphi\) abbreviates true \(\varphi\); and always abbreviates false \(R \varphi\) or equivalently \(\neg \bigcirc \neg \varphi\). Given a finite trace \(\pi\), we inductively define when an LTL\(_f\) formula \(\varphi\) is satisfied at an instant \(i \in \mathbb{N}\), in symbols \(\pi, i \models \varphi\), as follows:

- \(\pi, i \models \phi\) iff \(0 \leq i \leq \text{length}(\pi)\) and \(\pi(i) \models \phi\);
- \(\pi, i \models \neg \varphi\) iff \(\pi, i \models \varphi\);
- \(\pi, i \models \varphi_1 \land \varphi_2\) iff \(\pi, i \models \varphi_1\) and \(\pi, i \models \varphi_2\);
- \(\pi, i \models \bigcirc \varphi\) iff \(0 \leq i < \text{length}(\pi) - 1\) and \(\pi, i + 1 \models \varphi\);
- \(\pi, i \models \varphi_1 U \varphi_2\) iff for some \(j\) s.t. \(1 \leq i \leq j < \text{length}(\pi)\), we have \(\pi, j \models \varphi_2\), and for all \(k, i \leq k < j\), we have \(\pi, k \models \varphi_1\);

LDL\(_f\) is a temporal logic as natural as LTL\(_f\), but with the full expressive power of Monadic Second-Order logic over finite traces. LDL\(_f\) is obtained by merging LTL\(_f\) with regular expressions (RE\(_f\)) through the syntax of the well-know logic of programs PDL, *Propositional Dynamic Logic* (Fischer and Ladner 1979; Harel 1984), but adopting a semantics based on finite traces. LDL\(_f\) is an adaptation of LDL introduced in (Vardi 2011), which, like LTL\(_f\), is interpreted over infinite traces. Formally, given a set of propositional symbols \(P\), LDL\(_f\) formulae are built as follows:
\[ \varphi ::= \text{tt} \mid \text{ff} \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \langle \rho \rangle \varphi \]
\[ \rho ::= \phi \mid \lnot \varphi \mid \rho_1 + \rho_2 \mid \rho_1 ; \rho_2 \mid \rho^* \]

where \( \text{tt} \) and \( \text{ff} \) denote respectively the true and the false \( \text{LDL}_f \) formula (not to be confused with the propositional formula \text{true} and \text{false}); \( \phi \) denotes propositional formulae over \( P \); \( \rho \) denotes path expressions, which are regular expressions over propositional formulae \( \phi \) over \( P \) with the addition of the test construct \( \lnot \varphi \)? typical of PDL and are used to insert into the execution path checks for satisfaction of additional \( \text{LDL}_f \) formulae; and \( \varphi \) stand for \( \text{LDL}_f \) formulae built by applying boolean connectives and the modal operators \( \langle \rho \rangle \varphi \) and \( [\rho] \varphi \). Where \( [\rho] \varphi \) is an abbreviation for \( \lnot (\rho) \lnot \varphi \). We also introduce the abbreviations \( \text{end} = [\text{true}]\text{ff} \) and \( \text{last} = (\text{true})\text{end} \).

Intuitively, \( \langle \rho \rangle \varphi \) states that, from the current step in the trace, there exists an execution satisfying the regular expression \( \rho \) such that its last step satisfies \( \varphi \), while \( [\rho] \varphi \) states that, from the current step, all executions satisfying the regular expression \( \rho \) are such that their last step satisfies \( \varphi \). Also, note that given a regular expression \( \rho \), the \( \text{LDL}_f \) formula \( \langle \rho \rangle \text{end} \) is semantically equivalent to it.

Given a finite trace \( \pi \), we inductively define when an \( \text{LDL}_f \) formula \( \varphi \) is satisfied at an instant \( i \in \mathbb{N} \), in symbols \( \pi, i \models \varphi \), as follows:

\[ \pi, i \models \text{tt} \]
\[ \pi, i \models \lnot \varphi \text{ iff } i \not\models \varphi \]
\[ \pi, i \models \varphi_1 \land \varphi_2 \text{ iff } i \models \varphi_1 \land i \models \varphi_2 \]
\[ \pi, i \models \langle \rho \rangle \varphi \text{ iff } \exists j \geq i \text{ s.t. } (i, j) \in \mathcal{R}(\rho, \pi) \land j \models \varphi \]

where the relation \( \mathcal{R}(\rho, \pi) \) is defined inductively as follows:

- \( \mathcal{R}(\phi, \pi) = \{(i, i) \mid i \leq \text{length}(\pi) \land \pi(i) = \text{true}\} \)
- \( \mathcal{R}(\neg \varphi, \pi) = \{(i, i) \mid i \models \varphi\} \)
- \( \mathcal{R}(\rho_1 + \rho_2, \pi) = \mathcal{R}(\rho_1, \pi) \cup \mathcal{R}(\rho_2, \pi) \)
- \( \mathcal{R}(\rho_1 ; \rho_2, \pi) = \{(i, j) \mid \exists k \text{ s.t. } (i, k) \in \mathcal{R}(\rho_1, \pi) \land (k, j) \in \mathcal{R}(\rho_2, \pi)\} \)
- \( \mathcal{R}(\rho^*, \pi) = \{(i, k) \mid (i, j) \in \mathcal{R}(\rho, \pi) \land (k, j) \in \mathcal{R}(\rho^*, \pi) \land k > i\} \)

**From LTL\(_f\) to LDL\(_f\).** It is easy to encode LTL\(_f\) into LDL\(_f\): we can define a translation function \( tr \) defined by induction on the \( \text{LDL}_f \) formula as follows:

\[ tr(\phi) = \langle \phi \rangle \text{tt} (\phi \text{ propositional}) \]
\[ tr(\lnot \varphi) = \neg tr(\varphi) \]
\[ tr(\varphi_1 \land \varphi_2) = tr(\varphi_1) \land tr(\varphi_2) \]
\[ tr(O \varphi) = (\text{true})(tr(\varphi) \land \neg \text{end}) \]
\[ tr(\varphi_1 U \varphi_2) = ((tr(\varphi_1); \text{true}^*) (tr(\varphi_2) \land \neg \text{end})) \]

It is also easy to encode regular expressions, used as a specification formalism for traces into LDL\(_f\): \( \rho \) translates to \( \langle \rho \rangle \text{end} \). With \( \text{nnf} \), where \( \varphi \) is either an LTL\(_f\) or LDL\(_f\) formula, we mean the function that transforms \( \varphi \) by pushing negation inside until it is just used in front of atomic propositions, by applying the duality of the operators.

**Automata theory.** A deterministic finite automaton (DFA) (Rabin and Scott 1959) \( A \) is a tuple \( (Q, \Sigma, q_0, \delta, F) \) where \( Q \) is a finite set of states, \( \Sigma \) is the alphabet, \( q_0 \) is the initial state, and \( F \subseteq Q \) is the set of accepting states. \( \delta : Q \times \Sigma \to Q \) is the transition relation. A nondeterministic finite automaton (NFA) is defined as the DFA except for \( \delta \), which becomes a relation rather than a function, i.e. \( \delta \subseteq Q \times \Sigma \times Q \). An alternating finite automaton (AFA) (Chandra and Stockmeyer 1976; Vardi 1996) is defined as DFA and NFA, except for \( \delta \) that is defined as \( \delta : Q \times \Sigma \to B^+(Q) \), where \( B^+(Q) \) is a set of positive boolean formulas whose atoms are states of \( Q \). By \( L(A) \) we mean the set of all traces over \( \Sigma \) accepted by an automaton \( A \). DFAs are closed under boolean operations. A DFA \( A \) can be minimized, obtaining another DFA with the minimum number of states, in such a way that they are semantically equivalent. It can be shown that if a DFA is minimal, it is unique for the language it accepts. The DFAs are also closed under the following operations: concatenation, Kleene closure, existential and universal projection. Due to lack of space, we do not specify other details on these operations, and how to compute the actual automata. Any other detail can be found in any textbook of automata theory (e.g. see (Hopcroft, Motwani, and Ullman 2006)).

**Compositional Translation**

In this section, we describe the technique inductively translate each basic LTL\(_f\)/LDL\(_f\) formula and operators over them into (minimal) DFAs. We call the technique “compositional” due to its focusing on smaller subproblems and in the successive composition of partial results. We provide direct transformations from LTL\(_f\) to automata; for what concerns LDL\(_f\), we apply the transformation rules explained in the “Preliminaries” section. Finally, we will provide theoretical analysis of the technique.

**The Technique**

In what follows, we describe the transformation for each elementary formula and operator of LDL\(_f\) into an equivalent DFA. The approach is “bottom-up”: it computes the DFA of the deepest subformulae, and combines the partial results depending on the LDL\(_f\) operator under transformation. This is in contrast with the previous techniques known in the literature that are “top-down”: they proceed from the root operator of the formula in order to compute the next states (see e.g. LDL\(_f\)NFA in (De Giacomo and Vardi 2013, 2015; Brafman, De Giacomo, and Patrizi 2018)).

**tt and ff:** the logical true formula \( \text{tt} \) is equivalent to a DFA with an unique accepting state and a loop that accepts all symbols (Figure 1a). In other words, it is the minimal automaton that accepts the language \( \Sigma^* \). Its dual, \( \text{ff} \), is the automaton of the empty language (Figure 1b).

\( \varphi \land \varphi_2, \varphi \lor \varphi_2 \) and \( \lnot \varphi \): The boolean operations over LDL\(_f\) formulae are processed with the corresponding boolean operations over automata. For conjunction and disjunction, we use the product construction with respectively conjunction or disjunction of states as accepting conditions; for negation, we use the complementation of automata. The
output of these operations might require a further minimization and completion step.

\( (\phi) \varphi \): the diamond formula with a propositional formula as regular expression is equivalent to the automaton in Figure 1c. With the empty trace, the run succeeds. Otherwise, the next input symbol of the trace is read; if it satisfies \( \phi \), then the run proceeds with the simulation of the automaton associated to \( \varphi \) (starting from the state labelled with \( A_\varphi \)), else the run fails and goes to the sink state. Observe that the operation might require a further minimization step, even if \( A_\varphi \) is minimal; e.g. take \( \varphi = ff \) as example.

\( [\psi?] \varphi \) and \( [\psi?] \varphi \): The formulae can be reduced to \( \psi \land \varphi \) and \( \neg \psi \lor \varphi \), respectively.

\( (\rho_1 \cup \rho_2) \varphi \) and \( [\rho_1 \cup \rho_2] \varphi \): Both formulae are reducible to \( (\rho_1) \varphi \lor (\rho_2) \varphi \) and \( [\rho_1] \varphi \lor [\rho_2] \varphi \), respectively.

\( (\rho^*) \varphi \) and \( [\rho^*] \varphi \): It is enough to translate \( (\rho^*) \varphi \) and get the other by duality of the diamond operator, i.e. \( [\rho^*] \varphi \equiv \neg (\rho^*) \neg \varphi \). Hence, we will only consider \( (\rho^*) \varphi \). To compute the automaton \( A_{(\rho^*) \varphi} \), we first consider the case in which \( \rho \) does not contain any test. In this case, we have that the automaton \( A_\rho \) of \( \rho \) is equivalent to the automaton of \( (\rho)^{end} \), i.e. \( A_\rho = A_\rho^{(\rho)^{end}} \), as the semantics of LDLf formulae of the form \( (\rho)^{end} \) is the same of REf formulae \( \rho \). Hence, the automaton \( A_\rho^{(\rho)^{end}} \) can be computed using the well-known construction of DFA from regular expressions (See, e.g. (Hopcroft, Motwani, and Ullman 2006)). Then, we compute the Kleene closure of \( A_\rho \). Finally, we concatenate \( A_\rho \) and \( A_\rho \) to obtain the desired automaton. This approach can be generalized to handle tests as well in some cases, but not always, since it could happen that the verification of a test \( \psi \) could take more steps than the regular expression \( \rho \) itself. When this happens it is no longer true that \( A_\rho \) and \( A_\rho^{(\rho)^{end}} \) are equivalent since the presence of \( end \) in the second one would stop the evaluation of the test \( \psi \) too early, changing the semantics of the formula. Hence when we cannot guarantee that this does not happen, we simply fall back to using the classical algorithm that computes the DFA from \( (\rho^*) \varphi \) (De Giacomo and Vardi 2013; Brafman, De Giacomo, and Patrizi 2018), with the only difference that we recursively pre-compute the DFA \( A_\varphi \) for each test \( \psi \) and the DFA \( A_\varphi \) for \( \varphi \), and whenever we go to state \( \psi \) or \( \varphi \) in the DFA \( A_\varphi \) we actually go to the initial state of the DFAs \( A_\rho \) and \( A_\varphi \). Then we transform the DFA into a NFA as usual and then determinize it to obtain the desired DFA. The reason why we adopted two different approaches for \( (\rho^*) \varphi \) is that the case when \( \rho \) does not contain tests allows us to better decompose the problem. Intuitively, this happens because of the lack of universal transitions due to the absence of the test expressions in \( \rho \).

To summarize, in order to compute the DFA \( A_\varphi \) equivalent to an LDLf formula \( \varphi \), recursively apply the transformations stated above, one for each syntactic construct of the formula.

**Analysis**

Now we analyze the technique, proving correctness, termination, and running time complexity.

**Theorem 1. (Correctness)** Let \( \varphi \) be an LDLf formula and \( A_\varphi \) the corresponding DFA. Then for every TLTL interpretation \( \pi \) we have that \( \pi \models \varphi \iff \pi \in \mathcal{L}(A_\varphi) \).

**Proof.** We prove a more general statement, that is \( \forall \varphi, i \models \varphi \iff \pi(i, length(\pi)) \in \mathcal{L}(A_\varphi) \). Clearly, the claim of the theorem corresponds to the case \( i = 0 \). For \( i > 0 \), we proceed by induction on the structure of \( \varphi \).

\( \varphi = tt \). Then, on the one hand, \( \pi, i \models tt \). On the other hand, \( \pi(i, length(\pi)) \in \mathcal{L}(A_{tt}) \), where \( \mathcal{L}(tt) = \{ \pi^* \} \).

\( \varphi = ff \). Then, on the one hand, \( \pi, i \not\models ff \). On the other hand, \( \pi(i, length(\pi)) \not\in \mathcal{L}(A_{ff}) \), where \( \mathcal{L}(ff) = \emptyset \).

\( \varphi = \neg \psi \). Then, \( \pi, i \models \neg \psi \), and, by definition, \( \pi, i \not\models \varphi \).

By structural induction, we have that \( \pi(i, length(\pi)) \not\in \mathcal{L}(A_\varphi) \) and so \( \pi(i, length(\pi)) \not\in \mathcal{L}(A_{\neg \psi}) \), hence \( \pi(i, length(\pi)) \) is not accepted by \( A_\varphi = \overline{A_{\neg \psi}} \).

\( \varphi = \psi_1 \lor \psi_2 \). We have both \( \pi, i \models \psi_1 \) and \( \pi, i \models \psi_2 \).

By structural induction, we then have that \( \pi(i, length(\pi)) \in \mathcal{L}(A_{\psi_1}) \) and \( \pi(i, length(\pi)) \in \mathcal{L}(A_{\psi_2}) \), which is the condition of acceptance for \( \pi(i, length(\pi)) \) on \( A_{\psi_1} = \overline{A_{\psi_2}} \) or \( A_{\psi_2} = \overline{A_{\psi_1}} \).

\( \varphi = \psi_1 \land \psi_2 \). We have either \( \pi, i \models \psi_1 \) or \( \pi, i \models \psi_2 \).

By structural induction, we then have that \( \pi(i, length(\pi)) \in \mathcal{L}(A_{\psi_1}) \) or \( \pi(i, length(\pi)) \in \mathcal{L}(A_{\psi_2}) \), which is the condition of acceptance for \( \pi(i, length(\pi)) \) on \( A_{\psi_1} = \overline{A_{\psi_2}} \cup \overline{A_{\psi_2}} \).

\( \varphi = (\rho)^{\varphi} \). We proceed by induction on \( \rho \), and we show that for every \( \varphi' \), \( \pi, i \models (\rho)^{\varphi'} \iff \pi(i, length(\pi)) \in \mathcal{L}(A_{(\rho)^{\varphi'}}) \).

- \( \rho = \phi \). We have that \( \pi, i \models (\phi)^{\varphi'} \iff (i + 1) \in \mathcal{R}(\phi, \pi) \) and \( \pi, i + 1 \models \varphi \). Notice also that \( A_{(\phi)^{\varphi'}} \) is of the
form shown in Figure 1c. Observe that if \( i \geq \text{length}(\pi) \)
then \( \pi, i \models \langle \phi \rangle \varphi' \) is false, and indeed the empty trace
\( \pi(1, \text{length}(\pi)) = \varepsilon \) is not accepted by \( A_{\langle \phi \rangle \varphi'} \). If \( i < \text{length}(\pi) \),
then \( \pi, i \models \langle \phi \rangle \varphi' \) iff \( \pi(i) = \varepsilon \) and \( \pi, i + 1 \models \varphi' \),
which is iff the transition from \( q_0 \) and \( A_{\varphi'} \) is taken, and
then \( \pi(i + 1, \text{length}(\pi)) \in \mathcal{L}(A_{\varphi'}) \).

- \( \rho = \rho_1 \oplus \rho_2 \). Observe that \( \langle \rho_1 + \rho_2 \rangle \varphi' \equiv \langle \rho_1 \rangle \varphi' \lor \langle \rho_2 \rangle \varphi' \),
thus this case is addressed by applying the same reasoning as the one for conjunction.

- \( \rho = \rho_1 ; \rho_2 \). Observe that \( \langle \rho_1 ; \rho_2 \rangle \varphi' \equiv \langle \rho_1 \rangle \langle \rho_2 \rangle \varphi' \). By
induction on \( \rho_2 \) we have that \( \pi, i \models \langle \rho_2 \rangle \varphi' \iff \pi(1, \text{length}(\pi)) \in \mathcal{L}(A_{\langle \rho_2 \rangle \varphi'}) \). By induction on \( \rho_1 \),
we also observe that for all \( \psi \), \( \pi, i \models \langle \rho_1 \rangle \varphi' \iff \pi(1, \text{length}(\pi)) \in \mathcal{L}(A_{\langle \rho_1 \rangle \varphi'}) \). By replacing \( \psi \) with \( \langle \rho_2 \rangle \varphi' \),
and considering that the automaton \( A_{\langle \rho_1 ; \rho_2 \rangle \varphi'} \) is
by definition \( \mathcal{A}_{\langle \rho_1 \rangle \langle \rho_2 \rangle \varphi'} \), the thesis follows.

- \( \varphi = (\rho^*) \varphi' \). We first consider the case where \( \rho \) does not contain tests.
We prove this case by induction on \( n = \text{length}(\pi, i, \text{length}(\pi)) \).
First, assume \( n = 0 \). This implies that \( i \geq \text{length}(\pi) \),
and hence \( \pi(i, \text{length}(\pi)) = \varepsilon \), i.e. the empty trace.
Since we are out-of-bounds and no propositional formulæ
are executable, and the only case that matters is the
one with zero repetition of \( \rho \) in \( \pi^*; \pi, i \models (\rho^*) \varphi' \) holds iff \( \pi, i \models \varphi' \). By structural induction, \( \pi, i \models \varphi' \) holds iff \( A_{\varphi'} \) accepts \( \pi(i, \text{length}(\pi)) = \varepsilon \),
and now construct the
construction of \( A_{(\rho^*) \varphi'} \). It is the concatenation of \( A_{(\rho^*) \varphi \cap} \)
and \( A_{\varphi'} \). Since \( A_{\varphi'} \) accepts the empty trace by construction
(it is the Kleene closure of \( A_{\langle \rho \rangle \varphi \cap} \)), \( A_{(\rho^*) \varphi'} \) accepts
the empty trace iff \( A_{\varphi'} \) accepts the empty trace.
Now, assume that \( n > 0 \) and the claim holds for every
\( n' < n \). From the semantics of \( \langle \rho^* \rangle \varphi' \), we have that \( \pi, i \models \langle \rho^* \rangle \varphi' \) iff \( j \leq i \) s.t. \( \rho \) is a

It is also of interest to make some observations on the
intermediate automata generated by the technique. The computation of DFAs of simple formulæ \( tt, \langle \phi \rangle \varphi \) and \( \langle \phi \rangle \varphi \),
given the DFA for \( \varphi \), can be done in constant time, since
they don’t depend directly on the size of \( \phi \) nor \( \varphi \). Negation
consists in changing accepting states to rejecting states
and vice versa. The other boolean operations are translated
using products of DFAs, which are polynomial. The computation
of \( A_{(\rho \varphi) \varphi} \) without the occurrence of the \( * \) operator
can be handled reducing recursively to the previous cases
without introducing any non-determinism. The occurrence of the
\( * \) instead prevents us to reduce to the previous cases,
and introduces non-determinism due to the Kleene closure
and the concatenation operations, and hence exponential steps
to determinize the resulting automaton (Maslov 1970; Yu
Zhuang, and Salomaa 1994). More precisely, let us consider
a sub-formula \( \langle \rho^* \rangle \varphi \). If \( \rho \) does not contain tests and does
not contain star operators, then computing the DFA \( \mathcal{A}_{(\rho \varphi) \varphi} \) is
polynomial, and computing the DFA for the Kleene closure
\( \mathcal{A}_{(\rho \varphi) \varphi} \), is exponential w.r.t. the size of \( A_{(\rho \varphi) \varphi} \). As it
is exponential doing the concatenation with \( A_{\varphi} \), but w.r.t.
the size of \( A_{\varphi} \), hence the total contribution is one exponential. If
\( \rho \) contains star operators, then for the arguments above those
sub-expressions already contribute with an arbitrary number
of exponentials, and the outermost star contributes with
another exponential for the same arguments. If \( \rho \) contains
complex tests, then we switch to the AFA construction which contributes
with a double-exponential cost due to transformation to NFA
and to determinization to obtain the DFA.

Summarizing, any nested star operation gives, in the worst
number of exponential blow-up and hence is nonelementary.
Although this may sound discouraging, we observe that practical tools like Mona (Henrikson et al. 1995)
are nonelementary; yet, they perform very well in practice. We show
that also our implementation of the technique is competitive with
Mona and other tools. Also, observe that in our implementation,
like in Mona, we aggressively minimize the partial
DFA obtained after each compositional step. Since
the cost of DFA minimization for automata with explicit-state
representations can be done in \( \mathcal{O}(n \log n) \) (Hopcroft 1971),
this does not worsen the complexity of the technique, while

\footnote{Or we are guaranteed that the test is completed within the part of the word scanned by \( \rho \).}
in practice enhances it substantially because often the minimal DFA obtained from an NFA is of size comparable to the NFA itself, instead of being exponential in it.

In any case, since the technique is correct (c.f., Theorem 1), by the uniqueness of minimal DFAs, the returned DFA (once minimized) is at most double-exponentially larger than the LDL_f formula (De Giacomo and Vardi 2013, 2015; Braffman, De Giacomo, and Patrizi 2018).

Implementation

We have implemented the technique described in the previous section in a tool called Lydia³. Lydia is able to parse LTL_f and LDL_f in a grammar defined by us, and represents the syntactic tree using n-ary trees. It uses the Mona DFA library (Henriksen et al. 1995; Klarlund, Møller, and Schwartzbach 2001) to represent DFAs and perform operations over them. Note that we don’t use other Mona features related to the MSO logic parsing and manipulation. LydiaSynt is the extension of Lydia that also uses the Syft+ tool to perform LTL_f/LDL_f synthesis. Syft+ is an enhanced version of Syft, that enables dynamic variable ordering, used by Bansal et al. That is, after the computation of the MONA-based DFA, the program passes it to the Syft+ tool in order to compute the winning-set.

Semi-symbolic Automata Representation. Let \( A \) be a DFA over the alphabet \( 2^P \), where \( P \) is a set of \( k \) atomic propositions. Note that such alphabet is isomorphic to \( B^k \), where the vector \( v \in B^k \) identifies a subset \( \Pi \subseteq P \), such that the bit \( v_i \) is true iff \( p_i \in \Pi \). Due to exponential size of the alphabet in the number of propositional symbols \( |P| \), it is crucial to adopt a concise representation of automata transitions. To achieve this goal, we leverage the Mona DFA library for automata construction and manipulation. In Mona, the transitions of a DFA are symbolically represented as a shared multi-terminal binary decision diagram (shMBDD), where the transition relation of a DFA is encoded as a binary decision diagram (BDD) with multiple terminal nodes. The alphabets of these DFAs are the sets of bit vectors of length \( k \), i.e. \( B^k \), for some \( k \). In our case, each bit is associated to an atomic proposition appearing in the LDL_f formula. In addition to a compact representation on transitions of DFAs, the Mona DFA library provides efficient implementations of standard automata operations. These operations include product, (existential) projection, determination, and minimization. We extended the library so to include the Kleene closure, the concatenation, and the universal projection.

Existential and Universal Projections. In Mona, the existential projection of the \( i \)th bit (\( 1 \leq i \leq k \)), and the determination of its result, denoted as EPROJECT\((A, i)\) converts a DFA \( A \) recognizing a language \( L \) to a DFA \( A' \) recognizing the language \( L' \) where \( L' \) is the existential projection over bit \( i \) of \( L \). The process consists of removing the \( i \)th track of the MBDD and determining the resulting MBDD via on-the-fly subset construction. The universal projection, denoted as UPROJECT\((A, i)\), is also based on the subset

³The source code of Lydia can be found at https://github.com/whitemech/lydia
the representation of the alternation in the alphabet, through
the addition of universal and existential bits. This exploits
the asymmetry of the Mona DFA implementation, which is
symbolic in the transitions and explicit in the states. Hence,
it is less costly to add a transition rather than a state. More-
over, this also gives the opportunity to minimize the result-
ing DFA, hence saving computational resources for the fol-
lowing projections and determinizations.

A crucial difference with respect to the classic LDL\textsubscript{f}-to-
AFA transformation is that, whenever one of the atom occur-
rences we come across is either a test expression \( \psi \) or \( \varphi \),
instead of expanding those nodes as if they were states of
the AFA, we concatenate the current state to their DFAs. This
gives a good amount of compositionality also to this case,
which translates into more opportunity to minimize the par-
tial results, and hence in achieving greater performances.

**Heuristics.** As mentioned, we adopt aggressive minimiza-
tion after every step of the technique. Also, whenever the
technique starts computing a product between \( n \) automata,
we keep a priority queue to get the next two smallest
operands; the idea is to delay state blow-up of the partial
automaton as much as possible. This is a heuristics already
adopted by Bansal et al. and it is crucial for better scalability.

**Experimental Evaluation**

The evaluation has been designed to compare the perfor-
mance of Lydia and LydiaSynt against their respective
existing tools and approaches: Mona and Lisa for LTL\textsubscript{f}-to-
DFA conversion, and Syft+ and Lisa for synthesis. Both
LTL\textsubscript{f}-to-DFA conversion tools and synthesis tools are com-
pared on runtime and number of benchmarks solved within a
given timeout. We conduct our experiments on a benchmark
suite curated from prior works, spanning classes of realistic
and synthetic benchmarks: random conjunctions (400 cases)
(Zhu et al. 2017), single counters (20 cases), double coun-
ters (10 cases) etc. and Nim games (24 cases) (Tabajara and
Vardi 2019; Bansal et al. 2020) More details on each class
can be found in the supplementary material. In the case of
Lydia, the input LTL\textsubscript{f} formula is parsed and translated into
an LDL\textsubscript{f} formula. All experiments were conducted on a sin-
gle laptop equipped with an Intel Core i7-8665U CPU run-
ning at 1.90GHz with 16 GB of RAM.

**Comparison with Syft+.** Lydia has always better run-
times than Mona/Syft+, for DFA construction and there-
fore for the overall synthesis running time. This suggests
that working directly on LTL\textsubscript{f}/LDL\textsubscript{f} syntax, rather than pass-
ing first through MSO or FO and then to DFA, gives better
performances. This can be seen in particular for the DFA
construction runtime for single counter (Figure 2) and
doube counter benchmarks (Figure 3) and, for what concerns
synthesis, in Figure 4, where the Mona-based approach, i.e.
Syft+, is never better than LydiaSynt, especially on the
Nim benchmark.

**Comparison with Lisa.** We observe that Lydia is often
better than Lisa. That suggests that for the explicit part of
Lisa, going fully compositional is a better idea. In fact, the
assumption that LTL\textsubscript{f} formulae are conjunctions of multi-
ple smaller subformulae might not hold in some cases, es-
pecially outside synthesis domains. This can be seen in the
running times for the DFA construction on Nim benchmark
(Table 1), the cactus plot in Figure 5, and in the first part
of the running time of single-counter (Figure 2) and double-
counter (Figure 3). However, we have to remark that for
the last benchmarks of both the single and double counter, Lisa
and Lisa-explicit manage to construct the DFA, whereas
Lydia fails due to memout errors. This is due to different
approaches in the computation of the DFA product: Whilst
Lydia uses only the Mona DFA library, Lisa relies on
Mona for the computation of each subautomaton and then
combines them with SPOT (Duret-Lutz et al. 2016). More-
over, since Lisa implements a hybrid approach, it is able to
choose adaptively the right approach. Nevertheless, as the
cactus plot in Figure 5 shows, Lydia yields better run-

![Figure 2: DFA construction. Runtime for single-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.](image1)

![Figure 3: DFA construction. Runtime for double-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.](image2)
the problem is too large, also Lydia suffers from the state-space explosion, whereas Lisa are able to manage such inputs, thanks to their symbolic representation. Consequently, LydiaSynt suffers from the same limitations of Syft+.

A crucial thing to keep in mind is that Lydia processes the $\text{LTL}_f$ formula by translating it into $\text{LDL}_f$ and operating over it. Despite working on a more expressive logic formalisms, the overall performances are very good. That suggests this approach is pretty promising, and we believe that using direct transformations rules from $\text{LTL}_f$ to DFA would give us even better performances.

**Conclusions**

We proposed a fully compositional translation from $\text{LTL}_f/\text{LDL}_f$ to DFA. We do the transformation to DFA *directly* exploiting the structure of the formula, while previous work either relied on MSO/FO encoding, or they went through the computation of AFA. Moreover, we have empirically showed the advantages on the practical side. Indeed, Lydia and LydiaSynt are competitive with state-of-the-art tools for $\text{LTL}_f/\text{LDL}_f$-to-DFA translation and $\text{LTL}_f$ synthesis. Also, to the best of our knowledge, ours is the first work that provides a scalable and performant tool for the translation of $\text{LDL}_f$ to DFAs, and so also for $\text{LTL}_f$ synthesis, thanks to the integration with Syft+. As a future work, we would like to provide direct transformations from $\text{LTL}_f$ syntax to DFA, extend the approach to the pure-past versions of $\text{LTL}_f$ and $\text{LDL}_f$ (De Giacomo et al. 2020b), and improve the current implementation, in particular by implementing more advanced simplification rules for the input formulae and by exploiting an hash-consing data structure so to avoid to compute multiple times the same sub-automaton, including taking into account signature equivalences between formulae (e.g. see (Klarlund, Møller, and Schwartzbach 2001) about the DAG construction).
References


De Giacomo, G.; Stasio, A. D.; Fuggitti, F.; and Rubin, S. 2020b. Pure-Past Linear Temporal and Dynamic Logic on Finite Traces. In IJCAI.


Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2006. Introtd. to Automata Theory, Languages, and Computation.


Rabin, M. O.; and Scott, D. 1959. Finite automata and their decision problems. IBM J of Res. and Dev. 3: 114–125.


