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 (LDL_f) on finite traces into Deterministic Finite Automata DFA. That is, we inductively transform each $\text{LTL}_f/\text{LDL}_f$ subformula 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 LTL_f but also for full LDL_f.

Introduction

Linear Temporal Logics over finite traces (LTL_f), and its extension with regular expression, Linear Dynamic Logic (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 φ, we can build a deterministic finite automaton (DFA) $\mathcal{A}_\varphi$, whose alphabet is the set of propositional interpretations $\mathcal{P}$ of φ, 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 $\mathcal{A}_\varphi$ can be double-exponentially larger than the original formula φ. 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; Klarlund 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 LTL_f to DFA, proposed by (Zhu et al. 2017). Their tool `Syft` encodes 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 LTL_f synthesis. A more recent work (Bansal et al. 2020) proposed a hybrid approach to the problem of DFA construction from LTL_f formulae: first, they decompose the outermost conjunction in φ, where φ 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 φ_i into DFAs $\mathcal{A}_{\varphi_i}$ in explicit-state representation using `Mona`. Finally, they start doing the product between all the automata $\mathcal{A}_{\varphi_i}$; 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_f 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 deteminization 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, intersecion, 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-toDFA 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 $\pi = \pi_0, \dots, \pi_n$ of elements from the alphabet $2^{\mathcal{P}}$, containing all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We denote the length of the trace π as $length(\pi) \doteq n + 1$, and with $last(\pi) \doteq n$ the last index. We denote as $\pi(i) \doteq \pi_i$ 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 $\pi(i,j) \doteq \pi_i, \pi_{i+1}, \dots, \pi_{j-1}$ the segment of the trace π starting at the i-th step and ending at the j-th step (excluded). If $j > length(\pi)$ then $\pi(i,j) = \pi(i, length(\pi))$. 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 $\mathcal{P}$ of propositional symbols, LTL_f formulae are built as follows:

$$\varphi \quad ::= \quad \phi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \circ\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

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

- $\pi, i \models \phi$ iff $0 \leq i \leq length(\pi)$ and $\pi(i) \models \phi$;
- $\pi, i \models \neg\varphi$ iff $\pi, i \models \varphi$;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \circ\varphi$ iff $0 \leq i < length(\pi) - 1$ and $\pi, i+1 \models \varphi$;
- $\pi, i \models \varphi_1 \,\mathcal{U}\, \varphi_2$ iff for some j s.t. $1 \leq i \leq j < length(\pi)$, 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, is interpreted over infinite traces. Formally, given a set of propositional symbols $\mathcal{P}$, LDL_f formulae are built as follows:

$$\varphi \ ::= \ tt \mid ff \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\rho\rangle\varphi$$
$$\rho \ ::= \ \phi \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1;\rho_2 \mid \rho^*$$

where tt and ff denote respectively the true and the false LDL$_f$ formula (not to be confused with the propositional formula $true$ and $false$); ϕ denotes propositional formulae over $\mathcal{P}$; ρ denotes path expressions, which are regular expressions over propositional formulae ϕ over $\mathcal{P}$ with the addition of the test construct $\varphi?$ typical of PDL and are used to insert into the execution path checks for satisfaction of additional LDL$_f$ formulae; and φ stand for 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 $\neg\langle\rho\rangle\neg\varphi$. We also introduce the abbreviations $end = [true]ff$ and $last = \langle true\rangle end$.

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

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

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

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

- $\mathcal{R}(\phi,\pi) = \{(i,i+1) \mid i \leq length(\pi) \wedge \pi(i) \models \phi\}$
- $\mathcal{R}(\varphi?,\pi) = \{(i,i) \mid \pi, i \models \varphi\}$
- $\mathcal{R}(\rho_1 + \rho_2,\pi) = \mathcal{R}(\rho_1,\pi) \cup \mathcal{R}(\rho_2,\pi)$
- $\mathcal{R}(\rho_1;\rho_2,\pi) = \{(i,j) \mid \exists k \text{ s.t. } (i,k) \in \mathcal{R}(\rho_1,\pi) \wedge (k,j) \in \mathcal{R}(\rho_2,\pi)\}$
- $\mathcal{R}(\rho^*,\pi) = \{(i,i)\} \cup \{(i,j) \mid (i,k) \in \mathcal{R}(\rho,\pi) \wedge (k,j) \in \mathcal{R}(\rho^*,\pi) \wedge k > i\}$

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 LTL$_f$ formula as follows:

$$tr(\phi) = \langle\phi\rangle tt \ (\phi \text{ propositional})$$
$$tr(\neg\varphi) = \neg tr(\varphi)$$
$$tr(\varphi_1 \wedge \varphi_2) = tr(\varphi_1) \wedge tr(\varphi_2)$$
$$tr(\circ\varphi) = \langle true\rangle(tr(\varphi) \wedge \neg end)$$
$$tr(\varphi_1 \,\mathcal{U}\, \varphi_2) = \langle(tr(\varphi_1)?;true)^*\rangle(tr(\varphi_2) \wedge \neg end)$$

It is also easy to encode regular expressions, used as a specification formalism for traces into LDL$_f$: ρ translates to $\langle\rho\rangle end$. With $nnf(\varphi)$, where φ is either an LTL$_f$ or LDL$_f$ formula, we mean the function that transforms φ 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) $\mathcal{A}$ is a tuple $(Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ 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 δ, 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 δ 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 $\mathcal{L}(\mathcal{A})$ we mean the set of all traces over Σ accepted by an automaton $\mathcal{A}$. DFAs are closed under boolean operations. A DFA $\mathcal{A}$ 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

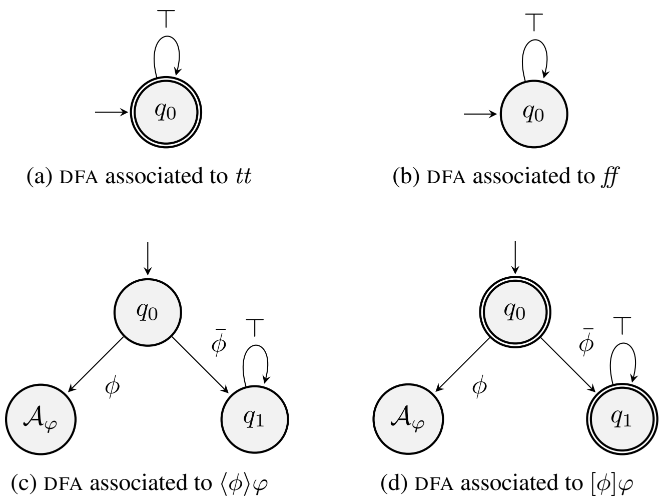
In this section, we describe the technique inductively translate each basic LTL$_f$/LDL$_f$ formula and operators over them into (minimal) DFAs. We call the technique "compositional" due to its focusing on smaller subproblems and in the successive composition of partial results. We provide direct transformations from LDL$_f$ to automata; for what concerns LTL$_f$, 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$2NFA in (De Giacomo and Vardi 2013, 2015; Brafman, De Giacomo, and Patrizi 2018)).

tt and ff: the logical true formula 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 Σ^*. Its dual, ff, is the automaton of the empty language (Figure 1b).

$\boldsymbol{\varphi_1 \wedge \varphi_2}$**,** $\boldsymbol{\varphi_1 \vee \varphi_2}$ **and** $\boldsymbol{\neg\varphi}$: 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



(a) DFA associated to tt (b) DFA associated to ff

(c) DFA associated to $\langle\phi\rangle\varphi$ (d) DFA associated to $[\phi]\varphi$

Figure 1: DFAs of elementary LDL_f formulae.

output of these operations might require a further minimization and completion step.

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

$[\boldsymbol{\phi}]\boldsymbol{\varphi}$: the box formula with a propositional formula as regular expression is equivalent to the automaton in Figure 1d. With the empty trace, the run succeeds. Otherwise, the next input symbol of the trace is read; if it satisfies ϕ, then the run proceeds with the simulation of the automaton associated to φ (starting from the state labelled with $\mathcal{A}_\varphi$), else the run succeeds and goes to the sink accepting state. Observe that the operation might require a further minimization step, even if $\mathcal{A}_\varphi$ is minimal; e.g. take $\varphi = tt$ as example.

$\langle\boldsymbol{\psi?}\rangle\boldsymbol{\varphi}$ **and** $[\boldsymbol{\psi?}]\boldsymbol{\varphi}$: The formulae can be reduced to $\psi \wedge \varphi$ and $\neg\psi \vee \varphi$, respectively.

$\langle\boldsymbol{\rho_1;\rho_2}\rangle\boldsymbol{\varphi}$ **and** $[\boldsymbol{\rho_1;\rho_2}]\boldsymbol{\varphi}$: Both formulae are reducible to $\langle\rho_1\rangle\langle\rho_2\rangle\varphi$ and $[\rho_1][\rho_2]\varphi$, respectively.

$\langle\boldsymbol{\rho_1+\rho_2}\rangle\boldsymbol{\varphi}$ **and** $[\boldsymbol{\rho_1+\rho_2}]\boldsymbol{\varphi}$: These formulae can be reduced to $\langle\rho_1\rangle\varphi \vee \langle\rho_2\rangle\varphi$ and $[\rho_1]\varphi \wedge [\rho_2]\varphi$, respectively.

$\langle\boldsymbol{\rho^*}\rangle\boldsymbol{\varphi}$ **and** $[\boldsymbol{\rho^*}]\boldsymbol{\varphi}$: It is enough to translate $\langle\rho^*\rangle\varphi$ and get the other by duality of the diamond operator, i.e. $[\rho^*]\varphi \equiv \neg\langle\rho^*\rangle\neg\varphi$. Hence, we will only consider $\langle\rho^*\rangle\varphi$. To compute the automaton $\mathcal{A}_{\langle\rho^*\rangle\varphi}$, we first consider the case in which ρ does not contain any test. In this case, we have that the automaton $\mathcal{A}_\rho$ of ρ, is equivalent to the automaton of $\langle\rho\rangle end$, i.e. $\mathcal{A}_\rho = \mathcal{A}_{\langle\rho\rangle end}$, as the semantics of LDL_f formulae of the form $\langle\rho\rangle end$ is the same of RE_f formulae ρ. Hence, the automaton $\mathcal{A}_{\langle\rho\rangle end}$ 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 $\mathcal{A}_\rho$, $\mathcal{A}_{\rho^*}$. Finally, we concatenate $\mathcal{A}_{\rho^*}$ and $\mathcal{A}_\varphi$ 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 ρ itself. When this happens it is no longer true that $\mathcal{A}_\rho$ and $\mathcal{A}_{\langle\rho\rangle end}$ 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 AFA from $\langle\rho^*\rangle\varphi$ (De Giacomo and Vardi 2013; Brafman, De Giacomo, and Patrizi 2018), with the only difference that we recursively pre-compute the DFA $\mathcal{A}_\psi$ for each test $\psi?$ and the DFA $\mathcal{A}_\varphi$ for φ, and whenever we go to state $\psi?$ or φ in the AFA of $\langle\rho^*\rangle\varphi$ we actually go to the initial state of the DFAs $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ Then we transform the AFA into a NFA as usual and then determinize it to obtain the desired DFA. The reason why we adopted two different approaches for $\langle\rho^*\rangle\varphi$ is that the case when ρ 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 ρ.

To summarize, in order to compute the DFA $\mathcal{A}_\varphi$ equivalent to an LDL_f formula φ, 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 φ be an LDL_f formula and $\mathcal{A}_\varphi$ the corresponding DFA. Then for every LTL_f -interpretation π we have that $\pi \models \varphi \iff \pi \in \mathcal{L}(\mathcal{A}_\varphi)$.*

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

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

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

$\varphi = \neg\varphi'$. Then, $\pi, i \models \varphi'$, and, by definition, $\pi, i \not\models \varphi$. By structural induction, we have that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ and so $\pi(i, length(\pi)) \notin \mathcal{L}(\mathcal{A}_{\neg\varphi})$, hence $\pi(i, length(\pi))$ is not accepted by $\mathcal{A}_\varphi = \overline{\mathcal{A}_{\varphi'}}$.

$\varphi = \varphi_1 \wedge \varphi_2$. We have both $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$. By structural induction, we then have that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_1})$ and $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_2})$, which is the condition of acceptance for $\pi(i, length(\pi))$ on $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cap \mathcal{A}_{\varphi_2}$.

$\varphi = \varphi_1 \vee \varphi_2$. We have either $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$. By structural induction, we then have that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_1})$ or $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi_2})$, which is the condition of acceptance for $\pi(i, length(\pi))$ on $\mathcal{A}_\varphi = \mathcal{A}_{\varphi_1} \cup \mathcal{A}_{\varphi_2}$.

$\varphi = \langle\rho\rangle\varphi'$. We proceed by induction on ρ, and we show that for every φ', $\pi, i \models \langle\rho\rangle\varphi' \iff \pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle\varphi'})$.

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

- $\rho = \psi?$. Observe that $\langle\psi?\rangle\varphi' \equiv \psi \wedge \varphi'$, thus this case is addressed by applying the same reasoning as the one for conjunction.
- $\rho = \rho_1 + \rho_2$. Observe that $\langle\rho_1 + \rho_2\rangle\varphi' \equiv \langle\rho_1\rangle\varphi' \vee \langle\rho_2\rangle\varphi'$, thus this case is addressed by applying the same reasoning as the one for disjunction.
- $\rho = \rho_1; \rho_2$. Observe that $\langle\rho_1;\rho_2\rangle\varphi' \equiv \langle\rho_1\rangle\langle\rho_2\rangle\varphi'$. By induction on ρ_2 we have that $\pi, i \models \langle\rho_2\rangle\varphi' \iff \pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho_2\rangle\varphi'})$. By induction on ρ_1, we have that for all ψ, $\pi, i \models \langle\rho_1\rangle\psi \iff \pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho_1\rangle\psi})$. By replacing ψ with $\langle\rho_2\rangle\varphi'$, and considering that the automaton $\mathcal{A}_{\langle\rho_1;\rho_2\rangle\varphi'}$ is by definition $\mathcal{A}_{\langle\rho_1\rangle\langle\rho_2\rangle\varphi'}$, the thesis follows.
- $\varphi = \langle\rho^*\rangle\varphi'$. We first consider the case where ρ does not contain tests. We prove this case by induction on $n = length(\pi(i, length(\pi)))$.
  First, assume $n = 0$. This implies that $i \geq length(\pi)$, and hence $\pi(i, length(\pi)) = \epsilon$, i.e. is the empty trace. Since we are out-of-bounds and no propositional formulae can be executed, and the only case that matters is the one with zero repetition of ρ in ρ^*: $\pi, i \models \langle\rho^*\rangle\varphi'$ holds iff $\pi, i \models \varphi'$. By structural induction, $\pi, i \models \varphi'$ holds iff $\mathcal{A}_{\varphi'}$ accepts $\pi(i, length(\pi)) = \epsilon$. Now, consider the construction of $\mathcal{A}_{\langle\rho^*\rangle\varphi'}$. It is the concatenation of $\mathcal{A}_{\langle\rho^*\rangle end}$ and $\mathcal{A}_{\varphi'}$. Since $\mathcal{A}_{\rho^*}$ accepts the empty trace by construction (it is the Kleene closure of $\mathcal{A}_{\langle\rho\rangle end}$), $\mathcal{A}_{\langle\rho^*\rangle\varphi'}$ accepts the empty trace iff $\mathcal{A}_{\varphi'}$ 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 exists $j \geq i$ s.t. either $j = i$ and $\pi, j \models \varphi'$ or there exists $j > i$ such that $(i, k) \in \mathcal{R}(\rho, \pi)$ and $(k, j) \in \mathcal{R}(\rho^*, \pi)$ and $\pi, j \models \varphi'$, with $k > i$. We want to prove that for every φ', $\pi, i \models \langle\rho^*\rangle\varphi'$ iff $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle\varphi'})$. We distinguish two cases; one in which there are zero repetitions of ρ ($j = i$), and the other when there are one or more ($j > i$).
  In case there are zero repetitions, we have that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle\varphi'})$ iff $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ by construction, and that $\pi, i \models \langle\rho^*\rangle\varphi'$. iff $\pi, i \models \varphi'$ by the semantics, so now we need to prove that $\pi(i, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ iff $\pi, i \models \varphi'$, but this is true by structural induction.
  In the other case, $j > i$, we have one or more repetitions of ρ. By construction, there exists a $k > i$ such that $\pi(i, k) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle end})$, $\pi(k, j) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle end})$, and $\pi(j, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$. We have that $\pi(i, k) \in \mathcal{L}(\mathcal{A}_{\langle\rho\rangle end})$ iff $(i, k) \in \mathcal{R}(\rho, \pi)$ by construction, $\pi(k, j) \in \mathcal{L}(\mathcal{A}_{\langle\rho^*\rangle end})$ iff $(k, j) \in \mathcal{R}(\rho^*, \pi)$ by induction on the length of the trace, and $\pi(j, length(\pi)) \in \mathcal{L}(\mathcal{A}_{\varphi'})$ iff $\pi, j \models \varphi'$ by structural induction. Combining the above equivalences, we get the thesis.

  Let us now consider the case in which ρ instead contains tests. Let $\psi_1?, \ldots, \psi_n?$ be all tests in ρ. Let $\mathcal{A}_{\psi_i}$ be the DFA associated to the test $\psi_i?$. Note that both these DFAs as well as $\mathcal{A}_{\varphi'}$ are correct by structural induction. Then we compute the AFA $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$ as in (De Giacomo and Vardi 2013; Brafman, De Giacomo, and Patrizi 2018), but with the difference that states of the form $\psi_i?$ or φ' are replaced by the initial states of $\mathcal{A}_{\psi_i}$ and $\mathcal{A}_{\varphi'}$, respectively. Moreover, the other states and transitions of these DFAs are added to the states and transitions of $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$. Then, we have that $\pi, i \models \langle\rho^*\rangle\varphi\ \pi(i, length(\phi)) \in \mathcal{L}(\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'})$, and since from $\mathcal{A}^{alt}_{\langle\rho^*\rangle\varphi'}$ we can obtain an equivalent DFA we get the thesis. □

It is also of interest to make some observations on the intermediate automata generated by the technique. The computation of DFAs of simple formulae *tt*, *ff*, $\langle\phi\rangle\varphi$ and $[\phi]\varphi$, given the DFA for φ, can be done in constant time, since they don't depend directly on the size of ϕ nor φ. 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 $\mathcal{A}_{\langle\rho\rangle\varphi}$ 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 ρ does not contain tests[2] and does not contains star operators, then computing the DFA $\mathcal{A}_{\langle\rho\rangle end}$ is polynomial, and computing the DFA for the Kleene closure, $\mathcal{A}_{\langle\rho^*\rangle end}$, is exponential w.r.t the size of $\mathcal{A}_{\langle\rho\rangle end}$. As it is exponential doing the concatenation with $\mathcal{A}_\varphi$, but w.r.t. the size of $\mathcal{A}_\varphi$ hence the total contribution is one exponential. If ρ 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 ρ contains 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 case, an exponential blow-up and hence is nonelementary. Although this may sound discouraging, we observe that practical tools like `Mona` (Henriksen 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

[2] Or we are guaranteed that the test is completed within the part of the word scanned by ρ.

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; Brafman, De Giacomo, and Patrizi 2018).

Implementation

We have implemented the technique described in the previous section in a tool called `Lydia`[3]. `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 represents 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 exension 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 $\mathcal{A}$ be a DFA over the alphabet $2^{\mathcal{P}}$, where $\mathcal{P}$ is a set of k atomic propositions. Note that such alphabet is isomorphic to $\mathbb{B}^k$, where the vector $v \in \mathbb{B}^k$ identifies a subset $\Pi \subseteq \mathcal{P}$, 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 $|\mathcal{P}|$, 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. $\mathbb{B}^k$, 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, determinization, 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 ith bit ($1 \le i \le k$), and the determinization of its result, denoted as $\text{EPROJECT}(\mathcal{A}, i)$ converts a DFA $\mathcal{A}$ recognizing a language L to a DFA $\mathcal{A}'$ recognizing the language L' where L' is the existential projection over bit i of L. The process consists of removing the ith track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction. The universal projection, denoted as $\text{UPROJECT}(\mathcal{A}, i)$, is also based on the subset construction used by the existential one, however, while in the existential projection the acceptance is true iff *exists* a state in the subset that is accepting, whereas in the universal projection the acceptance is true iff *all* the states in the subset are accepting. These two operations will be important building blocks for other operations.

Concatenation and Kleene Closure. The technique presented in the previous section requires adding nondeterministic transitions to the DFA operands of certain operations. In the case of concatenation between two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, the nondeterministic choice is made in the accepting states of $\mathcal{A}_1$, F_1, because these states should behave as if they were the initial state of $\mathcal{A}_2$; this can be implemented by adding all the transitions that leave the initial state of $\mathcal{A}_2$, q_0^2, to the states of $\mathcal{A}_1$ in F_1. Analogously, in the case of the Kleene closure of $\mathcal{A}$, the accepting states F should additionally behave as if they were the initial state of $\mathcal{A}$, q_0, and it can be implemented by adding all the transitions that leave the accepting state q_0 to the states in F.

In general, the new transitions might render the automaton nondeterministic, which cannot be handled by `Mona` APIs. Instead, we first add an auxiliary fresh bit e to the alphabet, and in the states where the nondeterministic choice happens, we use e to resolve the non-determinism, so to make the transitions deterministic. For example, in the concatenation described above, the transitions from each $f \in F_1$ that belongs to $\mathcal{A}_1$ will have the bit e set to true, whereas the new transitions will have the bit set to false, $\bar{e}$; similarly, in the Kleene closure, the old transitions will have the bit e set to true, and the new transitions will have it set to false $\bar{e}$. This will ensure that the result of those operations is still a DFA, as $e\phi_1 \wedge \bar{e}\phi_2 = \bot$, with ϕ_1 and ϕ_2 being propositional formulae. Finally, the desired automaton is $\mathcal{A} = \text{EPROJECT}(\mathcal{A}', i_e)$, where i_e is the index of the bit e.

Construction of the AFA. When we handle the case of $\langle \rho^* \rangle \varphi$ and $\rho \not\equiv \langle \rho \rangle end$, because of tests, we need to resort to constructing an AFA, and then determinizing it. Since `Mona` does not provide constructs for a direct implementation of AFAs, we rely on building intermediate DFAs, similarly to what has been done for concatenation and Kleene closure, on a bigger alphabet having two types of auxiliary bits: existential, to be projected via EPROJECT (as before) and *universal*, to be projected via UPROJECT. Let q be the current state to expand in the computation of the AFA, and let ϕ_q the formula over Q that determines the next transitions. Without loss of generality, assume ϕ_q is in disjunctive normal form, and assume that each clause is indexed across all the products and each atom occurrence is indexed within its clause. Let us call such indices i and j, respectively. Then, the construction adds a transition for each atom occurrence (i.e. an AFA state), whose guard is determined by the AFA transformation rules (De Giacomo and Vardi 2013; Brafman, De Giacomo, and Patrizi 2018) in conjunction with the instantiation of the existential and universal bits, corresponding to the binary representation of the indices i and j, respectively.

Intuitively, to obtain the DFA corresponding to the AFA, instead of doing the subset construction on-the-fly (which would need to keep track of *sets of sets* of states), we push

[3]The 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. Moreover, this also gives the opportunity to minimize the resulting DFA, hence saving computational resources for the following projections and determinizations.

A crucial difference with respect to the classic LDL_f-to-AFA transformation is that, whenever one of the atom occurrences we come across is either a test expression $\psi?$ or φ, 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 partial results, and hence in achieving greater performances.

Heuristics. As mentioned, we adopt aggressive minimization 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 performance of `Lydia` and `LydiaSynt` against their respective existing tools and approaches: `Mona` and `Lisa` for LTL_f-to-DFA conversion, and `Syft+` and `Lisa` for synthesis. Both LTL_f-to-DFA conversion tools and synthesis tools are compared 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 counters (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_f formula is parsed and translated into an LDL_f formula. All experiments were conducted on a single laptop equipped with an Intel Core i7-8665U CPU running at 1.90GHz with 16 GB of RAM.

Comparison with `Syft+`. `Lydia` has always better runtimes than `Mona`/`Syft+`, for DFA construction and therefore for the overall synthesis running time. This suggests that working directly on LTL_f/LDL_f syntax, rather than passing 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 double 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_f formulae are conjunctions of multiple smaller subformulae might not hold in some cases, especially 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). Moreover, 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-

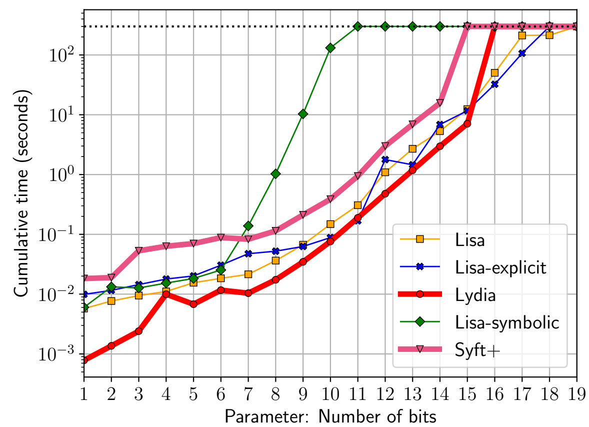


Figure 2: DFA construction. Runtime for single-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.

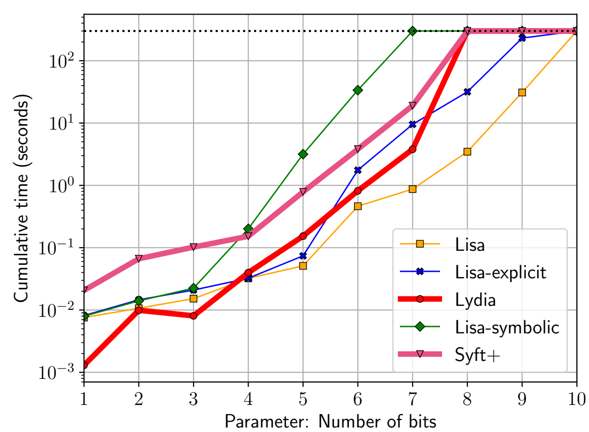


Figure 3: DFA construction. Runtime for double-counter benchmarks. Plots touching black line means time/memout. Timeout is at 300 sec.

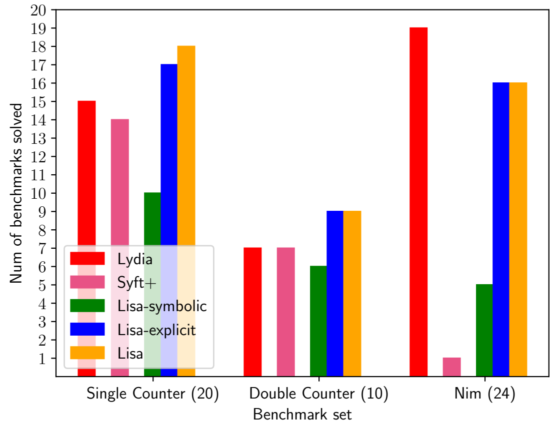


Figure 4: Number of benchmarks synthesized from each non-random benchmark class. Each benchmark has a timeout of 300 seconds.

Benchmark					
Name	Lydia	Mona-based	Lisa-expl.	Lisa-symb.	Lisa
`nim_1_1`	**0.01**	0.15	0.07	0.07	0.07
`nim_1_2`	**0.02**	—	0.15	0.16	0.16
`nim_1_3`	**0.05**	—	0.07	1.43	0.06
`nim_1_4`	**0.09**	—	0.14	267.23	0.13
`nim_1_5`	**0.17**	—	0.27	—	0.25
`nim_1_6`	**0.30**	—	0.63	—	0.54
`nim_1_7`	**0.54**	—	1.20	—	1.02
`nim_1_8`	**0.82**	—	1.87	—	1.83
`nim_2_1`	**0.05**	—	0.14	1.49	0.10
`nim_2_2`	**0.20**	—	0.84	—	0.81
`nim_2_3`	**1.47**	—	4.95	—	4.95
`nim_2_4`	**7.00**	—	26.07	—	24.33
`nim_2_5`	**34.86**	—	125.56	—	108.86
`nim_2_6`	**114.87**	—	—	—	—
`nim_2_7`	—	—	—	—	—
`nim_2_8`	—	—	—	—	—
`nim_3_1`	**0.40**	—	3.15	—	2.67
`nim_3_2`	**9.93**	—	84.34	—	78.31
`nim_3_3`	**142.16**	—	—	—	—
`nim_3_4`	—	—	—	—	—
`nim_4_1`	**8.97**	—	110.10	—	109.79
`nim_4_2`	—	—	—	—	—
`nim_5_1`	**243.62**	—	—	—	—
`nim_5_2`	—	—	—	—	—

Table 1: Running time (in seconds) for DFA construction on the Nim benchmark set. In bold the minimum running time for a given benchmark. — means time/memout. Timeout at 300 sec.

ning times than `Lisa` in the majority of cases (given the timeout of 300 seconds for each benchmark). In Figure 4, `LydiaSynt` shows to be competitive with state-of-the-art synthesis tools like `Lisa`. However, unsurprisingly, when

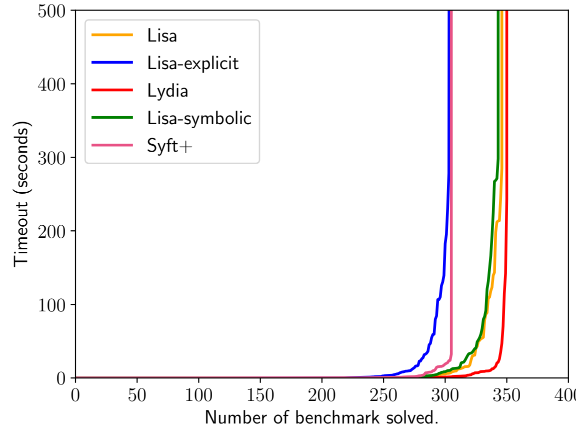


Figure 5: DFA construction. Cactus plot indicating number of benchmarks each tool can solve for a given timeout. Each benchmark whose running time was greater than 300 seconds was counted as ∞.

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 LTL$_f$ formula by translating it into 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 LTL$_f$ to DFA would give us even better performances.

Conclusions

We proposed a fully compositional translation from LTL$_f$/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 LTL$_f$/LDL$_f$-to-DFA translation and 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 LDL$_f$ to DFAs, and so also for LDL$_f$ synthesis, thanks to the integration with `Syft+`. As a future work, we would like to provide direct transformations from LTL$_f$ syntax to DFA, extend the approach to the pure-past versions of LTL$_f$ and 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).

Acknowledgments

This work is partially supported by the ERC Advanced Grant WhiteMech (No. 834228) and by the EU ICT-48 2020 project TAILOR (No. 952215).

References

Bacchus, F.; Boutilier, C.; and Grove, A. 1996. Rewarding Behaviors. In *AAAI*, 1160–1167.

Bacchus, F.; and Kabanza, F. 1998. Planning for temporally extended goals. *Ann. Math. Artif. Intel.* 5–27.

Bansal, S.; Li, Y.; Tabajara, L.; and Vardi, M. 2020. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *AAAI*, 9766–9774.

Brafman, R.; De Giacomo, G.; and Patrizi, F. 2018. LTLf/LDLf Non-Markovian Rewards 1771–1778.

Brafman, R. I.; and De Giacomo, G. 2019a. Planning for LTLf/LDLf Goals in Non-Markovian Fully Observable Nondeterministic Domains. In *IJCAI*.

Brafman, R. I.; and De Giacomo, G. 2019b. Regular Decision Processes: A Model for Non-Markovian Domains. In *IJCAI*, 5516–5522.

Camacho, A.; Baier, J. A.; Muise, C. J.; and McIlraith, S. A. 2018. Finite LTL Synthesis as Planning. In *ICAPS*, 29–38.

Camacho, A.; Icarte, R. T.; Klassen, T. Q.; Valenzano, R. A.; and McIlraith, S. A. 2019. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. In *IJCAI*, volume 19, 6065–6073.

Camacho, A.; and McIlraith, S. A. 2019. Strong Fully Observable Non-Deterministic Planning with LTL and LTLf Goals. In *IJCAI*, 5523–5531.

Chandra, A. K.; and Stockmeyer, L. J. 1976. Alternation. In *FOCS*, 98–108.

De Giacomo, G.; Favorito, M.; Iocchi, L.; Patrizi, F.; and Ronca, A. 2020a. Temporal Logic Monitoring Rewards via Transducers. In *KR*, 860–870.

De Giacomo, G.; Iocchi, L.; Favorito, M.; and Patrizi, F. 2019. Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications. In *ICAPS*, 128–136.

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

De Giacomo, G.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*, 854–860.

De Giacomo, G.; and Vardi, M. Y. 2015. Synthesis for LTL and LDL on Finite Traces. In *IJCAI*, 1558–1564.

De Giacomo, G.; and Vardi, M. Y. 2016. LTLf and LDLf Synthesis Under Partial Observability. In *IJCAI*, IJCAI'16, 1044–1050.

Duret-Lutz, A.; Lewkowicz, A.; Fauchille, A.; Michaud, T.; Renault, E.; and Xu, L. 2016. Spot 2.0—A Framework for LTL and ω-Automata Manipulation. In *ATVA*, 122–129.

Fischer, M. J.; and Ladner, R. E. 1979. Propositional dynamic logic of regular programs. *JCSS* 194–211.

Fogarty, S.; Kupferman, O.; Vardi, M. Y.; and Wilke, T. 2015. Profile trees for Büchi word automata, with application to determinization. *Information and Computation* .

Gerevini, A. E.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *AIJ* 173(5-6): 619–668.

Harel, D. 1984. Dynamic Logic. In *"Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic"*, 497–604.

Henriksen, J. G.; Jensen, J.; Jørgensen, M.; Klarlund, N.; Paige, R.; Rauhe, T.; and Sandholm, A. 1995. *Mona: Monadic second-order logic in practice*, 89–110.

Hopcroft, J. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, 189–196.

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

Klarlund, N. 1997. Mona & Fido: The logic-automaton connection in practice. In *CSL*, 311–326.

Klarlund, N.; Møller, A.; and Schwartzbach, M. I. 2001. *MONA Implementation Secrets*, 65–89.

Maslov, A. 1970. Estimates of the number of states of finite automata. In *Doklady Akademii Nauk*, 1266–1268.

Pešić, M.; Bošnački, D.; and van der Aalst, W. M. 2010. Enacting declarative languages using LTL: avoiding errors and improving performance. In *SPIN*, 146–161.

Pnueli, A. 1977. The temporal logic of programs. In *SFCS*, 46–57.

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

Tabajara, L. M.; and Vardi, M. Y. 2019. Partitioning Techniques in LTLf Synthesis. In *IJCAI*, 5599–5606.

Tabakov, D.; and Vardi, M. Y. 2005. Experimental evaluation of classical automata constructions. In *LPAR*, 396–411.

Vardi, M. 2011. *The rise and fall of linear time logic*. URL http://www.cs.rice.edu/~vardi/papers/gandalf11-myv.pdf.

Vardi, M. Y. 1996. *An automata-theoretic approach to linear temporal logic*, 238–266. Springer Berlin Heidelberg.

Yu, S.; Zhuang, Q.; and Salomaa, K. 1994. The state complexities of some basic operations on regular languages. *TCS* 315–328.

Zhu, S.; Tabajara, L. M.; Li, J.; Pu, G.; and Vardi, M. Y. 2017. Symbolic LTLf Synthesis. In *IJCAI*, 1362–1369.

Zhu, S.; Tabajara, L. M.; Pu, G.; and Vardi, M. Y. 2020. On the Power of Automata Minimization in Temporal Synthesis. *arXiv preprint arXiv:2008.06790* .