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

Giuseppe De Giacomo and Marco Favorito

Sapienza University of Rome
{lastname@diag.uniroma1.it}

Introduction

- Linear Temporal Logics over finite traces (LTL_f), and its extension with regular expression, Linear Dynamic Logic (LDL_f) [1], are important logic formalisms extensively used in Artificial Intelligence and Computer Science.
- 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 [1]. 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.
- In this work, we propose a fully compositional approach to handle both LTL_f formulae and LDL_f formulae. 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 $\text{LTL}_f/\text{LDL}_f$ 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 $\text{LTL}_f/\text{LDL}_f$ and automata theory relied on the transformation of $\text{LTL}_f/\text{LDL}_f$ formulae into Alternating Automata on finite words (AFA), which can be eventually transformed into Nondeterministic Finite Automata (NFA), and in turn determinized into DFAs [1]. 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 [2, 3]. On the practical side, we provide an implementation that employs such a compositional technique, and showing its competitiveness with existing tools [4, 5]. Our tool can be used both for $\text{LTL}_f/\text{LDL}_f$-toDFA construction, and as a $\text{LTL}_f/\text{LDL}_f$ 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.

How it works

Here 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".

- **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 1). In other words, it is the minimal automaton that accepts the language Σ^*. Its dual, ff, is the automaton of the empty language (Figure 2).
- **$\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi$**: 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 output of these operations might require a further minimization and completion step.
- **$\langle\phi\rangle\varphi$**: the diamond formula with a propositional formula as regular expression is equivalent to the automaton in Figure 3. 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 $\varphi = ff$ as example.
- **$[\phi]\varphi$**: the box formula with a propositional formula as regular expression is equivalent to the automaton in Figure 4. 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.
- **$\langle\psi?\rangle\varphi$ and $[\psi?]\varphi$**: The formulae can be reduced to $\psi \wedge \varphi$ and $\neg\psi \vee \varphi$, respectively.
- **$\langle\rho_1;\rho_2\rangle\varphi$ and $[\rho_1;\rho_2]\varphi$**: Both formulae are reducible to $\langle\rho_1\rangle\langle\rho_2\rangle\varphi$ and $[\rho_1][\rho_2]\varphi$, respectively.
- **$\langle\rho_1+\rho_2\rangle\varphi$ and $[\rho_1+\rho_2]\varphi$**: 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.
- **$\langle\rho^*\rangle\varphi$ and $[\rho^*]\varphi$**: 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. [6]). 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 $\langle\rho^*\rangle\varphi$ [1, 7], 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.

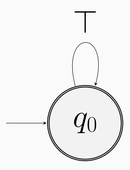

Fig. 1: DFA associated to tt

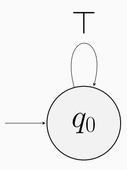

Fig. 2: DFA associated to ff

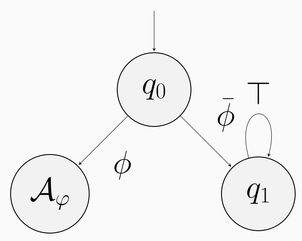

Fig. 3: DFA associated to $\langle\phi\rangle\varphi$

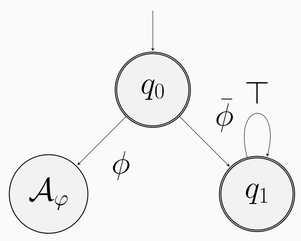

Fig. 4: DFA associated to $[\phi]\varphi$

Experiments and Implementation

We have implemented the technique described in the previous section in a tool called `Lydia`. The source code of `Lydia` can be found at https://github.com/whitemech/lydia. `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 [5, 2] 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 $\text{LTL}_f/\text{LDL}_f$ 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.
In the paper you can find experimental resulsts showing the competitiveness with state-of-the-art tools `Mona`/`Syft+` and `Lisa`/`LisaSynt`.

References

[1] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *IJCAI*. 2013, pp. 854–860.
[2] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. "MONA Implementation Secrets". In: 2001, pp. 65–89.
[3] Shufang Zhu et al. "On the Power of Automata Minimization in Temporal Synthesis". In: *arXiv preprint arXiv:2008.06790* (2020).
[4] Suguman Bansal et al. "Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications". In: *AAAI*. 2020, pp. 9766–9774.
[5] Jesper G. Henriksen et al. "Mona: Monadic second-order logic in practice". In: 1995, pp. 89–110.
[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introd. to Automata Theory, Languages, and Computation*. 2006.
[7] Ronen Brafman, Giuseppe De Giacomo, and Fabio Patrizi. "LTLf/LDLf Non-Markovian Rewards". In: (2018), pp. 1771–1778.

Acknowledgements

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