DAG-Based Compositional Approaches for LTLf to DFA Conversions

Suguman Bansal
Georgia Institute of Technology, USA
suguman@gatech.edu

Yash Kankariya
Georgia Institute of Technology, USA
ykankariya3@gatech.edu

Yong Li
University of Liverpool, UK
yong.li3@liverpool.ac.uk

***Abstract*— Scalable and efficient conversions of LTL *over finite horizon* (LTLf) to their *deterministic finite automata* (DFA) remain a critical bottleneck in several applications of LTLf. Recently, *compositional approaches* have seen remarkable success in scaling the conversion to large formulas. Here the input formula is first decomposed into smaller subformulas, each of which can be easily converted to their DFAs, then these DFAs are composed to generate the desired DFA. This work proposes a series of simple-yet-effective optimizations to improve the performance of compositional approaches based on reducing the number of composition steps required to generate the desired DFA.**

We incorporate these optimizations in a tool called Lisa2 that builds on one of the state-of-the-art tools for LTLf-to-DFA conversion. A comprehensive empirical evaluation of Lisa2 demonstrates overall improvements on both parameters of the number of benchmarks solved and runtime. Most remarkably, it demonstrates significant improvement over structured benchmarks where runtime speedups range between 1.5x to 8000x under fair comparisons to prior state-of-the-art tools.

I. Introduction

Linear Temporal Logic over finite traces [1] (LTLf) is the finite-horizon counterpart of Linear Temporal Logic (LTL) over infinite traces [2]. LTLf is rapidly gaining popularity among real-world applications where behaviors are better expressed over a finite but unbounded horizon. These include applications in planning and synthesis [3], [4], [5], [6], reinforcement learning [7], [8], [9], business processes [10], verification [11].

A critical challenge facing its applications is the conversion of LTLf specifications into their equivalent *deterministic finite automata (DFAs)*. This is not unexpected since the LTLf-to-DFA conversion exhibits a double-exponential blow-up in the size of the input specification in the worst-case [1], [12]. Yet, state-of-the-art LTLf-to-DFA conversion tools like Lisa [13] and Lydia [14] often succeed at converting large formulas. Their success can be attributed to *compositional algorithms* which are split into two phases. First, in the *decomposition phase* a large LTLf formula is decomposed into smaller subformulas using the formula's *Abstract Syntax Tree* (AST). Next, in the *composition phase*, subformulas at the leaves of the AST are converted to their DFAs using direct LTLf-to-DFA tools suitable for smaller formulas, such as Spot [15] or Mona [16]. Then, these DFAs are composed using language-theoretic and/or automata-based operations to obtain the DFA of the original formula. For DFA composition, the AST of the formula is traversed bottom-up.

Through this work, we propose a series of simple yet effective optimizations to improve the performance of compositional algorithms. Our first optimization aims at striking a balance between the time spent in converting subformulas at the leaves of the AST into their DFAs and the time spent in composing the intermediate DFAs. The deeper the AST is unrolled, the smaller are the subformulas at the leaves of the AST. While these smaller subformulas may be easier to convert into their DFAs, it increases the number of composition steps required to traverse the AST. To this end, we propose to unroll the AST on their outermost boolean operators only. In contrast, both Lisa and Lydia unroll at the extremes: Lisa unrolls on the outermost conjunction only whereas Lydia unrolls completely till the propositional literals.

Our other optimizations focus on reducing the number of composition steps required during the bottom-up traversal by modifying the AST. Our first optimization is based on eliminating subformula duplication within the AST. This eliminates multiple computations of the DFA of the same subformula that may be present at multiple locations in the AST of a formula. Such duplication is not uncommon in structured, real-world formulas. Our second optimization is based on using semantics-preserving syntactic transformations to the formula. In particular, subformulas of the form $\phi = \bigwedge_{i=1}^{k}(\psi \vee \phi_i)$ are rewritten as $\phi = \psi \vee \bigwedge_{i=1}^{k} \phi_i$, as the latter requires fewer composition steps: The former representation of ϕ will require $2k-1$ composition steps (k steps to create the DFA for all $(\psi \vee \phi_i)$ and $k-1$ steps from the outer conjunction of these clauses). Whereas, the latter will require only k steps of which $k-1$ steps are required to create the DFA for the large conjunction and one more is required to compose with ψ. Neither Lisa nor Lydia incorporate either of these optimizations.

We have implemented these optimizations in a tool Lisa2 that builds on the existing tool Lisa. The compositional algorithm in Lisa2 differs from Lisa as follows: (a). In the decomposition phase, the formula is unrolled on all outermost boolean operations using the formula's AST, (b). Next, there is an additional *optimization phase* in which duplicate removal and syntactic transformations modify the AST to a DAG as opposed to the AST, (c). Finally, in the composition phase, formulas at the *leaves* of the DAG are converted to their DFA and then these intermediate DFAs are composed in a

bottom-up traversal of the DAG. Lisa2 also differs from Lisa and Lydia in supporting multiple representations for DFAs. It permits Spot's [15] labeled-graph and Reduced-Ordered BDD (ROBDD) [17] (which is used by Lisa) as well as Mona's Shared Multi-Terminal BDD (ShMTBDD) (which is used by Lydia) [16]. Permitting both datastructures makes Lisa2 more versatile than prior tools. An additional benefit is that this enables fair comparison with Lisa and Lydia. While these prior tools implement differing compositional approaches, a fair comparison of these algorithms has not been possible since the performance of these tools is also affected by the complementary strengths of the underlying datastructure for DFAs. In particular, ROBDD may be slower but require less memory whereas ShMTBDD can be blazingly fast but are memory exhaustive. With the flexibility in choice of DFA datastructure, Lisa2 can compare different algorithmic approaches by ensuring that their underlying datastructures are identical, hence rendering fairer comparisons.

A comprehensive empirical evaluation demonstrates significant improvements over prior state-of-the-art tools in both the number of benchmarks solved and their runtime. We evaluated the performance of Lisa2 against Lisa and Lydia on LTLf benchmarks (a collection of randomly generated formulas and structured formulas) from the LTLf track in SYNTCOMP2023[1]. While Lisa2 outperforms both baselines comprehensively, its performance on the structured benchmarks is most remarkable. Not only Lisa2 solves $\sim$50% more structured benchmarks than prior approaches, it also demonstrates runtime improvements in the range of 1.5x-8000x (with more benchmarks recording high runtime improvement), highlighting the strength of our tool on realistic benchmarks.

II. Preliminaries and Notations

A. Linear Temporal Logic over Finite Traces (LTLf)

LTLf [1] extends propositional logic with finite-horizon temporal operators. The syntax of LTLf over a finite set of propositions Prop is identical to LTL, and defined as

$$\varphi := \mathsf{true} \mid \mathsf{false} \mid a \in \mathsf{Prop} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{X}\varphi \mid \varphi_1 \mathrm{U} \varphi_2$$

where X (Next) and U (Until), are temporal operators. We include their dual operators, N (Weak Next) and R (Release), defined as $\mathrm{N}\varphi \equiv \neg\mathrm{X}\neg\varphi$ and $\varphi_1 \mathrm{R} \varphi_2 \equiv \neg(\neg\varphi_1 \mathrm{U} \neg\varphi_2)$. We also use typical abbreviations such as $\mathrm{F}\varphi \equiv \mathsf{true}\mathrm{U}\varphi$, $\mathrm{G}\varphi \equiv \mathsf{false}\mathrm{R}\varphi$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. The semantics of LTLf can be found in [1].

Wlog, we assume formulas are given in *negation normal form* (NNF), i.e., the negation operator ($\neg$) appears in front of propositions only. In the given syntax, all formulas can be converted to their NNF with no blow-up in length.

Every LTLf formula φ over Prop can be converted into a deterministic finite-state autoamta (DFA) D with alphabet $\Sigma = 2^{\mathsf{Prop}}$ and at most double-exponential number of states in $|\varphi|$ such that $\mathcal{L}(D) = \mathcal{L}(\varphi)$ [1].

[1]https://www.syntcomp.org

B. Abstract Syntax Tree

The *Abstract Syntax Tree (AST)* is an n-ary tree representation of an LTLf formula. Formally, for an LTLf formula ϕ, (1). Every node corresponds to a subformula of the formula φ. In particular, the root of the AST corresponds to the formula itself, (2). For every node, the *operator* of the node is given by the outermost (primary) operator of its subformula, (3). For every node, its children correspond to the immediate subformulas of the formula in that node. Formulas with unary operators (such as $\neg$, X, F, and so on), binary operators (U and R), and n-ary operators ($\vee$ and $\wedge$) consist of one, two, and n-many children, respectively. The order of children is crucial for temporal operators U and R. For the remaining operators, the order of children does not alter the semantics of the formula since the operators are commutative.

III. Related Work

Optimizations in compositional LTLf-to-DFA approaches.: The current state-of-the-art AST-based compositional tools, Lisa [13] and Lydia [14], employ various optimizations to counter its inherent non-elementary complexity. In addition to aggressive DFA minimization [16] and order in which DFAs at each node are composed [13], the tools optimize on the depth to which the ASTs are unrolled during formula decomposition. For instance, Lisa decomposes the original formula at its outermost conjunction only, thus creating a k-ary AST of depth one. Thus, the leaves represent subformulas as opposed to atomic propositions. On the other hand, Lydia [14] generates the full k-ary AST up to literals in the leaves. This way tools attempt to trade-off between resources spent in the direct conversion of subformulas at the leaves and composition steps. The tools also optimize on the data structure used for explicit state-space representations of the DFAs. Lisa stores DFAs as labeled-graphs and in symbolical state-space using *Reduced-Ordered Binary Decision Diagrams (ROBDD)* [17] if necessary. Lydia stores DFAs using *Shared Multi-Terminal Binary Decision Diagrams (ShMTBDD)* of Mona [16].

Direct LTLf-to-DFA conversions.: Spot [15] and Mona [16] are two popularly used tools for direct LTLf-to-DFA conversions of smaller formulas. Spot translates the LTLf formula into an LTL formula with equivalent semantics, converts this formula into a Büchi automaton [18], and then transforms this Büchi automaton into a DFA. The Mona-based approach translates LTLf into *first-order logic over finite traces (FOL)* and then Mona converts the FOL formula into a DFA. Both generate minimal DFAs in explicit state-space representation.

DAG-based compositional approaches.: Mona also uses directed acyclic graphs (DAGs) to represent formulas as we do in this work [19]. The differences lie in the ways we identify equivalent subformulas and the depth of the DAGs. Mona identifies equivalent formulas ϕ and ϕ' by checking whether there is an order-preserving renaming of propositions in ϕ such that ϕ and ϕ' are identical, while we check the syntax equivalence of two formulas, simpler but much more

efficient. Moreover, Mona's DAG unrolls till its literals while our DAG unrolls on boolean operators only. To the best of our knowledge, Lisa and Lydia do not use DAG since no intermediate DFAs are stored for later use in their source code.

Optimizations in LTL to automata conversion: The conversion of LTL (Linear Temporal Logic) [2] to automata forms has received much attention. While the conversion incurs a single-exponential and double-exponential blow-up for the non-deterministic and deterministic versions automata versions, significant work has gone into developing optimizations for LTL to automata. However, these optimizations are too sophisticated for LTLf to automata translations, where relatively simpler translations have been shown to be more effective.

IV. Optimizations

We propose a series of optimizations to be applied to compositional approaches for LTLf-to-DFA conversion. The first optimization (Section IV-A) determines the depth to which an input formula's AST is unrolled during formula decomposition into smaller subformulas. The remaining two optimizations are geared to reduce the computation required during the composition phase by reducing the number of composition operations. Here, the first optimization compresses this AST by removing duplicate subformulas (Section IV-B). The second optimization performs semantics-preserving syntactic transformations that are guaranteed to reduce the number of composition steps (Section IV-C).

A. Depth of AST Unrolling

Our first optimization is based on the depth to which an input LTLf formula is unrolled to obtain smaller subformulas. We propose to unroll subformulas only if their outermost operator is a boolean operator. Recall, since we assume formulas are given in NNF, the outermost boolean operators are effectively only the conjunction operator or the disjunction operator as negations appear before propositions only. Consequently, for formulas at the leaves of this tree, the outermost operator could be any of the temporal operators. Figure 1 illustrates such an unrolling of an AST.

We make this choice to strike a balance between the conversion of subformulas to DFAs at the leaves vs. the composition of DFAs at intermediate nodes to obtain the final DFA. Prior state-of-the-art approaches Lisa and Lydia take diametrically opposite routes in this regard. Lisa unrolls the AST at its outermost conjunction only. I.e., given an LTLf formula $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, Lisa decomposes the original formula into the n-subformulas given by φ_is. The disadvantage of this decomposition is that in the worst case, the φ_is could be too large to be handled by an off-the-shelf LTLf-to-DFA conversion tools like Spot or Mona. The advantage, however, is that once the DFAs for the φ_is are created, the approach requires only $n-1$ composition steps, where each composition consists of polynomial-time operations of DFA product and DFA minimization only. In contrast, Lydia unrolls the AST completely. I.e., its leaves comprise of literals (propositions or their negation). This ensures that the DFA at the leaf node is obtained trivially. However, not only do the number of composition operations increase dramatically, the composition operations may become more complex. In particular, the composition at nodes with a boolean operator comprise of polynomial-time operations identical to Lisa, but the composition at nodes with temporal operators may involve exponential-time operations such as projection and/or determinization.

Our choice to unroll only on boolean operators ensures that all composition operations require polytime operations only while also ensuring that the size of subformulas at the leaves are small, hence combining the benefits of Lisa and Lydia.

B. Duplicate Removal

For our next optimization, we observe that in several formulas, an intermediate subformula may appear more than once in the formula's AST. This results in redundant computation during the composition phase as it generates the DFA for equivalent subformulas multiple times. To eliminate such redundant recomputation, we propose to merge nodes of equivalent subformulas. This is illustrated in Figure 2 where formula θ_1 that appeared twice in Figure 1 has been merged into one node. Such duplication removal will result in the AST being converted to a DAG as the merged nodes are required to serve multiple parent nodes.

In order to merge nodes in an AST, we must check if the formulas at two or more nodes are equivalent. LTLf formula equivalence is PSPACE-complete, hence we must resort to computationally inexpensive approaches to identify formula equivalence. Tools such as Spot offer inexpensive syntactic checks to examine formula equivalence. We combine these checks with leveraging the parent-child relationship between nodes in the AST to identify equivalent formulas.

To elaborate further, first we identify formula equivalence between the leaf nodes of the AST using syntactic checks, and merge each class of equivalent formulas into one leaf node. This converts the AST into a DAG as the merged leaf nodes will now serve multiple parent nodes. Next, it is easy to see that two non-leaf nodes are equivalent if all their children nodes are identical and their operators are identical. All such formula equivalence in non-leaf nodes can be identified efficiently in a single bottom-up traversal of the DAG that simultaneously merges equivalent non-leaf nodes into one node.

Note that this procedure may fail to recognize equivalent subformulas that do not adhere to our inexpensive checks. Despite this incompleteness, we observe that sometimes it can reduce the number of nodes in the AST/DAG by 30-40% in negligible time, hence demonstrating its effectiveness.

C. Semantics-Preserving Transformation

The final optimization aims to reduce the number of composition steps using *semantics-preserving syntactic transformation* of the formula. Lemma 1 motivates our optimization:

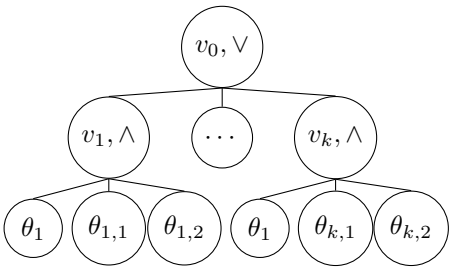


Fig. 1: AST unrolled on boolean operators only.

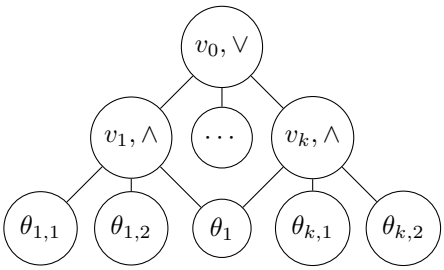


Fig. 2: After duplicate removal. θ_1 has been merged.

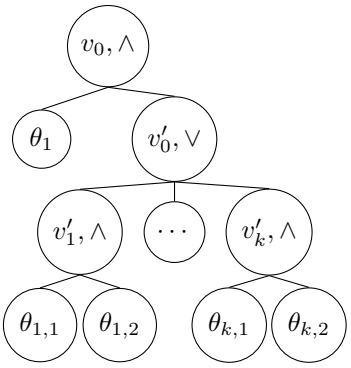


Fig. 3: After Transformation. θ_1 has been pulled out.

Lemma 1: Consider the following formula,

$$\phi = \circ'^{k}_{i=1}\Big((\theta_1 \circ \cdots \theta_l) \circ (\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big) \tag{1}$$

where $\circ, \circ' \in \{\vee, \wedge\}$ s.t. $\circ' \neq \circ$, and for all $i \in [k]$, $m_i \geq 0$.

Then ϕ is equivalently written as:

$$\phi' = (\theta_1 \circ \cdots \theta_l) \circ \Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big) \tag{2}$$

using the laws of associativity. Assuming the DFAs for all θ_i and $\theta_{i,j}$ are given, the required composition steps to create the DFA for ϕ is $\mathcal{O}(l \cdot (k-1))$ more than those required to create the DFA for ϕ'.

Proof: We begin with a sketch of the argument. In practice, a product over k DFAs requires $k-1$ products between two DFAs. Now, notice that the intermediate DFA for the common segment $\theta_c = (\theta_1 \circ \cdots \theta_l)$ is constructed $k-1$ times more in ϕ that in ϕ'. By pulling out the common segment θ_c using the laws of associativity in ϕ', the DFA for θ_c is constructed only once, amounting to the difference.

The formal argument follows: Let us first analyze the number of products required in ϕ. For every $i \in [k]$, the clause $(\theta_1 \circ \cdots \theta_l) \circ (\theta_{i,1} \circ \cdots \theta_{i,m_i})$ requires $l + m_i - 1$ products. Next, these clauses are combined using products to obtain ϕ. Since there are k clauses, we require $k-1$ additional products. Therefore, evaluating ϕ requires $(k-1) + \Sigma_{i=1}^{k}(l-1+m_i) = k \cdot l - 1 + \Sigma_{i=1}^{k} m_i$ product operations.

Next, we analyse the number of products required in ϕ'. For every $i \in [k]$, the clause $(\theta_{i,1} \circ \cdots \theta_{i,m_i})$ requires $m_i - 1$ products. In addition, these k clauses are combined using $k-1$ products to form the DFA for $\Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big)$. Hence, $\Big(\circ'^{k}_{i=1}(\theta_{i,1} \circ \cdots \theta_{i,m_i})\Big)$ requires $k - 1 + \Sigma_{i=1}^{k}(m_i - 1) = (\Sigma_{i=1}^{k} m_i) - 1$ operations. Combined with l many θ_js to obtain ϕ', we require $l - 1 + \Sigma_{i=1}^{k} m_i$ products to form ϕ'. Therefore, constructing the DFA via ϕ' requires $\mathcal{O}(l \cdot (k-1))$ fewer operations than creating the same DFA via ϕ, where l is the number of subformulas common to k-many clauses in ϕ. ■

Our optimization, therefore, applies the associative law to transform nodes of the form ϕ to nodes of the form ϕ' in the DAG obtained after duplicate removal. As earlier, the transformation is carried out by an analysis of the parent-child relations between nodes. A node v_0 is eligible for the transformation if the formula it represents is of the form ϕ, i.e. : (a) the outermost boolean operator should differ from the outermost boolean operator of all of its children, and (b) all its children share a common child of their own, referring to $\theta_c = (\theta_0 \circ \cdots \theta_l)$ in ϕ. In the DAG, the common child of all children is simply a common grandchild node. The common grandchildren are obtained from the intersection of all of children of v_i's for $i \geq 1$. In Figure 2, node v_0 is eligible for the transformation with a single common grandchild in θ. When eligible, the transformation *pulls out* all common segment θ_c. In the DAG, this translates to promoting all common grandchildren of v_0 to direct children of v_0 and the ealier children of v_0 are modified accordingly. Figure 2 to Figure 3 illustrate the transformation. Observe that the transformation may result in the creation of new nodes such as $v'_0, \cdots, v'_k$ in Figure 3.

As earlier, these transformations are carried out in a single bottom-up traversal (reverse topological order) of the DAG starting with the leaf nodes. In instances when the transformation results in the creation of a new node (such as $v'_0, v'_1, \cdots v'_k$ in Figure 3), the new nodes are examined for duplicates using the earlier approach. Then the transformation is recursively applied to v'_0 first and then to $v'_1, \cdots v'_k$ before returning to the next node as per the reverse topological order.

V. Compositional Algorithm

For the sake of completion, we present an outline of the compositional algorithm. W.l.o.g., our algorithm receives an LTLf formula in NNF and outputs a minimal DFA for the formula. The algorithm proceeds in three phases: First is the *decomposition phase* in which the input LTLf formula is decomposed into smaller subformulas based on its AST. The AST is unrolled on boolean operators only. This is followed by the *optimization phase* in which the proposed duplication removal and semantic-transformations are applied. As a result, the AST is converted to a DAG. Finally, in the *composition phase*, the subformulas at the *leaves* of the DAG, i.e. nodes with no outgoing edges in the DAG, are converted to their

minimal DFA form. Next, the DAG is traversed bottom-up starting with the leaves, i.e. the DAG is traversed in reverse topological order. During this traversal, the minimal DFA at a node is created if the minimal DFA at all its children have already been constructed. The primary difference from the AST-based composition is that the DFA at a node in the AST can be removed from memory as soon as the DFA in its parent node has been constructed. In the case of a DAG, the DFA at a node is discarded only after the DFA at *all* its parent nodes have been generated.

VI. Implementation Details

We have implemented compositional algorithm in a tool called Lisa2[2]. Lisa2 takes an LTLf formula in NNF as its input and outputs its minimal DFA in explicit representation.

In brief, Lisa2 extends a current state-of-the-art tool Lisa to incorporate the optimizations described in Section IV. In detail, Lisa2 has been written in C++. It uses Spot LTLf parser to parse the input formula. The input formula is decomposed into a DAG following the optimizations described in Section IV. To convert the subformulas at the leaves of the DAG, Lisa2 converts the LTLf formulas at the leaf nodes to their equivalent first-order logic (FOL) and uses Mona to convert the FOL formulas to their minimal DFAs. The DFAs are then composed as described in Section V. Similar to Lisa [13], Lisa2 deploys two performance-enhancing heuristics (a) *aggressive DFA minimization*, i.e. each DFA (intermediate or final) is minimized as soon as it is created, and (b) *smallest-first* heuristic that always picks the smallest two (minimal) DFAs to compose during a k-way product construction (for both conjunction and disjunction).

Lisa2 generates DFA in explicit-state representation, i.e., the states are given explicitly and the transitions are given as labeled formulas over the propositions of the input LTLf formula. Lisa2 supports two datastructures to represent the final and all intermediate DFAs: (a) Spot's labeled graphs and Reduced Ordered BDD (ROBDD) and (b). Mona's Shared Multi-Terminal BDD (ShMTBDD). We refer to these two variants of our tool as Lisa2-Spot and Lisa2-Mona, respectively. These tool variants use the DFA manipulation APIs provided by Spot and Mona, respectively, for all DFA operations including the product construction and minimization.

Tool Features: By supporting both Spot and Mona, Lisa2 is the only LTLf-to-DFA conversion tool that can support both datastructures, adding to its versatility in applications. Another motivation to support both DFA datastructures is to enable fair comparison for future algorithmic advances in LTLf-to-DFA conversion tools. Prior tools Lisa and Lydia support only one of the two Spot's labeled graphs + ROBDD combination and ShMTBDD, respectively. These datastructures have complementary benefits (ROBDD may be slower but require less memory whereas ShMTBDD are faster but are memory extensive.) and a bear significant impact the performance of their tools. As a result, performance comparisons between prior tools are unable to differentiate between the improvement caused by the algorithm vs. the improvement caused by the datastructure. Thus the ability to switch between DFA datastructures within Lisa2 creates the opportunity for more fair comparisons of algorithmic advances in LTLf-to-DFA tools.

[2]https://github.com/suguman-lab/lisa2

A. Implementation-Level Optimizations

Lisa2 incorporates several implementation-level optimizations. Few key ones are described below.

First, formulas of the form $\mathrm{G}(\bigwedge_{i=0}^{m}\phi_i)$ and $\mathrm{F}(\bigvee_{i=0}^{m}\phi_i)$ are equivalently written as $\bigwedge_{i=0}^{m}(\mathrm{G}\phi_i)$ and $\bigvee_{i=0}^{m}(\mathrm{F}\phi_i)$, respectively, to promote deeper decomposition on boolean operators. Had the formulas been retained in their earlier format, then the formulas would not be decomposed any further since the outermost operator is temporal. This optimization generates smaller subformulas.

Second, Lisa2 already identifies few subformula duplications (using Spot's inexpensive methods to determine formula equivalence) during the unrolling of the formula's AST. As a result, the outcome of the unrolling may already be a DAG as opposed to an AST. We do this as we observed that in some cases, the AST obtained from unrolling on boolean operators could become very large. Combining the unrolling with a shallow duplication-removal curb the growth in the AST.

We observe that in practice most DAG/AST nodes do not possess a common grandchild, making the node ineligible for the semantics-preserving transformation. Instead, it is more likely that several nodes possess a *popular* grandchild that may be a child of most but not all children of the node. In these cases, we perform the transformation only with the children that share the popular grandchild.

VII. Experimental Analysis

A. Design and Setup for Empirical Evaluation

Baselines and Fair Comparisons: We compare Lisa2 to the three state-of-the-art baselines: Lydia, Lisa, and Lisa-Explicit. All three tools are based on compositional algorithms. They differ in the depth of unrolling, few algorithmic details, and the underlying DFA datastructure. Lydia unrolls to the literals whereas Lisa and Lisa-Explicit unroll on the outermost conjunction only. In terms of DFA data structure, Lydia uses Mona's ShMTBDD while Lisa and Lisa-Explicit use Spot's labeled-graphs and ROBDDs. Since tool performance is known to be impacted by the DFA datastructure, we establish the following fair comparisons:

- Lisa2-Mona vs. Lydia
- Lisa2-Spot vs. Lisa and Lisa-Explicit

All tools accept inputs in Spot-parsable format, ensuring consistency among tools in input format.

Benchmarks: We use benchmarks from the LTLf-track at SYNTCOMP 2023[3]. We evaluate on 490 benchmarks of which 400 formulas are generated randomly and the remaining 90 are structured formulas derived from the "two-player games" category. Among the structured benchmarks, we use 20, 10, and 60 benchmarks from the single counter, double counter, and nim benchmark classes, respectively.

Set-up: All experiments were conducted on a single node of a high-performance cluster (https://pace.gatech.edu/). Each node consists of four quad-core Intel-Xeon processors running at 2.6 GHz with 4hrs timeout and 16GB of RAM each.

B. Performance-Related Observations

We begin by examining the performance of Lisa2 against its counterparts w.r.t. runtime and number of benchmarks solved. Overall, Lisa2 not only solves more benchmarks that all other counterparts, it also improves the runtime significantly. Most remarkable is its performance on the structured benchmarks where Lisa2 solves ∼50% more benchmarks and displays runtime improvements in the range of 2x-8000x. We describe our observations and inferences in detail below.

Lisa2 demonstrates the best overall performance: The cactus plots of the performance of all tools in Figure 4a (cactus plot on all benchmarks) and Figure 4b (cactus plot on structured benchmarks only) demonstrate that variants of Lisa2 solve the most number of benchmarks in both cases. Recall the fair comparisons from the previous section. We observe that on all benchmarks, Lisa2 Mona outperforms Lydia and Lisa2-Spot is comparable to/better than its fair counterparts.

Lisa2 comprehensively outperforms its fair counterparts on structured benchmarks. Lisa2-Spot solves almost twice as many benchmarks that its fair counterparts while Lisa2-Mona solves ∼40% more benchmarks than Lydia. This highlights the benefits of our optimizations on realistic benchmarks. More broadly, it reflects the merits of identifying and leveraging patterns appearing in structured (realistic) formulas.

Next, we examine each structured benchmark class in detail.

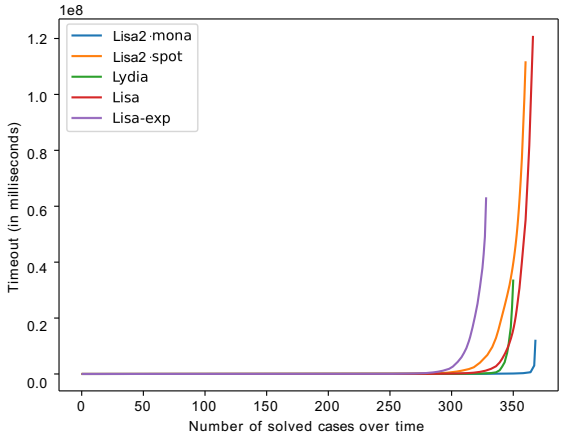
Lisa2 performs remarkably on the nim benchmarks.: Both variants of Lisa2 outperform their fair counterparts by a large margin in both, the number of benchmarks solved and runtime. Lisa2 solves almost twice as many benchmarks as their counterparts (Figure 5). Furthermore, the runtime speedup ranges between 2x-8000x with most benchmarks displaying greater than 500x speedup on Mona's ShMTBDD data structure; and on average 5x speedup on Spot's labeled-graph and ROBDD data structure (Table I).

This outcome is impressive as the nim-benchmarks had proven to be challenging for prior compositional approaches. This occurs since on these benchmarks the intermediate (minimal) DFAs tend to be very large even though final (mininal) DFA is quite small. Hence, it is not uncommon for Lisa or Lydia to fail at an intermediate stage due to memory or time shortage. We attribute Lisa2's success to our optimizations in reducing the number of compositional steps. For most benchmarks in the nim-class, after duplication removal and semantic transformations, the resulting DAG comprised of 5-20% fewer compositions steps than the AST obtaining from unrolling on boolean operators only. In another class of nim-benchmarks derived from [20], this reduction even ranged between 20-50%, resulting in better performance gain.

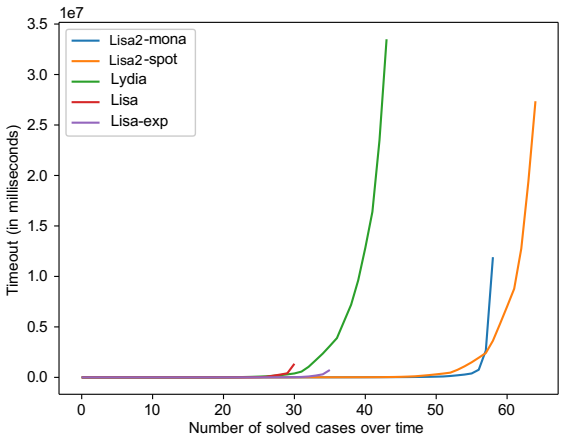
These experiments clearly demonstrate the advantage of reducing the composition steps.

Performance on counter benchmarks: On the single- and double-counter classes of benchmarks, Lisa2-Mona demonstrates 1.4x-100x runtime improvement over Lydia. Whereas, Lisa2-Spot displays 1.5x-100x runtime improvement over Lisa but is sometimes slower than Lisa-Explicit, as demonstrated in Table II.

We attribute the performance of Lisa2 on the counter



(a) Cactus plot: All benchmarks. Timeout 4hrs



(b) Cactus plot: Structured benchmarks. Timeout 4hrs

Fig. 4: Overall Performance

[3]SYNTCOMP: https://www.syntcomp.org. Benchmarks were taken from https://github.com/whitemech/finite-synthesis-datasets/tree/main. We chose benchmarks from the whitemech repository because (a). All SYNTCOMP23 benchmarks in LTLf track were obtained by converting the whitemech benchmarks to TLSF format, (b). All baseline tools natively support the format used in whitemech as opposed to the TLSF format used by SYNTCOMP.

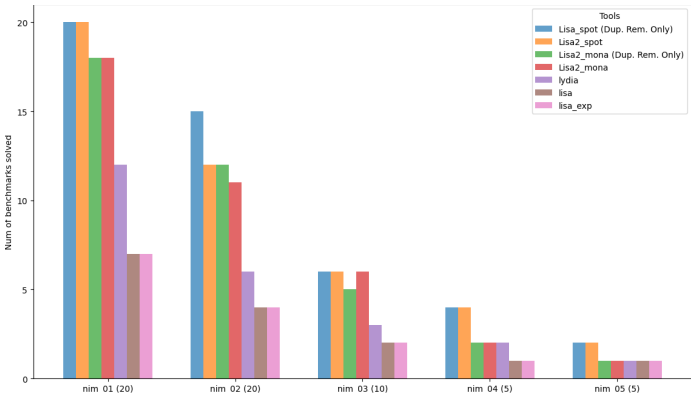


Fig. 5: Number of benchmarks solved on the nim class. x- and y-axes denote benchmark class (total instances in brackets) and num. of benchmarks solved, respectively.

Cases	Lisa2-Mona	Lydia	Lisa2-Spot	Lisa	Lisa-exp
nim_01_1	10	21	10	0	0
nim_01_2	30	44	20	0	0
nim_01_3	30	89	40	10	10
nim_01_4	50	277	50	20	10
nim_01_5	50	1087	80	50	40
nim_01_6	80	4413	100	120	100
nim_01_7	90	16929	110	210	220
nim_01_8	120	64115	160	-	-
nim_01_9	190	181994	200	-	-
nim_01_10	230	780581	240	-	-
nim_01_11	380	1652381	330	-	-
nim_01_12	420	3203658	360	-	-
nim_01_13	620	-	560	-	-
nim_01_14	820	-	780	-	-
nim_01_15	2220	-	910	-	-
nim_01_16	9780	-	810	-	-
nim_01_17	49100	-	810	-	-
nim_01_18	-	-	1300	-	-
nim_01_19	108890	-	1490	-	-
nim_01_20	-	-	1820	-	-
nim_02_1	20	69	30	20	10
nim_02_2	40	483	80	60	80
nim_02_3	80	5979	180	380	400
nim_02_4	130	92548	380	2040	1760
nim_02_5	200	650682	820	-	-
nim_02_6	350	3574672	2430	-	-
nim_02_7	480	-	8890	-	-
nim_02_8	1150	-	76890	-	-
nim_02_9	2540	-	348840	-	-
nim_02_10	3950	-	466940	-	-
nim_02_11	-	-	-	-	-
nim_02_12	17690	-	452700	-	-
nim_02_13	-	-	-	-	-
nim_02_14	-	-	-	-	-
nim_02_15	-	-	3970670	-	-
nim_03_1	60	465	110	160	160
nim_03_2	160	52220	810	3050	3260
nim_03_3	1110	1653251	27080	-	-
nim_03_4	2110	-	91610	-	-
nim_03_5	4000	-	396780	-	-
nim_03_6	7270	-	1215350	-	-
nim_04_1	180	23813	1050	3920	4520
nim_04_2	2540	7083597	75430	-	-
nim_04_3	-	-	1670340	-	-
nim_04_4	-	-	6634310	-	-
nim_05_1	1310	755274	23500	112100	123220
nim_05_2	-	-	1718010	-	-

TABLE I: Runtime in millisecs for nim. Timeout 4hrs

Cases	Lisa2-Mona	Lydia	Lisa2-Spot	Lisa	Lisa-exp
counter_1	10	7	10	10	10
counter_2	10	17	40	60	60
counter_3	10	33	310	560	540
counter_4	10	57	20	10	10
counter_5	20	82	20	30	10
counter_6	50	156	50	60	30
counter_7	200	371	170	300	100
counter_8	810	1111	690	9560	290
counter_9	3520	4212	2870	125350	990
counter_10	15190	16611	14770	113410	3700
counter_11	66390	76345	61690	-	15130
counter_12	366250	474631	277750	-	90000
counter_13	1910580	2419211	1762580	-	430210
counter_14	9241030	10013917	7970190	-	-
counter_15	-	-	-	-	550
counters_1	10	17	50	80	70
counters_2	10	55	10	10	10
counters_3	20	150	30	80	30
counters_4	80	1103	130	1220	170
counters_5	600	25784	1000	29540	1060
counters_6	7850	660391	8090	941940	7380
counters_7	74590	-	60060	-	47080

TABLE II: Runtime in millisecs for counters. Timeout 4hrs.

benchmarks to the unrolling depth. This is because for most of these benchmarks, duplication removal and semantic transformation did not result in any significant reduction in composition steps as the benchmarks exhibit neither multiple occurrences of a subformula nor are their patterns amenable to the syntactic transformation. We observed that Lydia would fail because of the accumulation in number and size of intermediate DFAs in its AST that unrolls till the literals. This is aggravated by the ShMTDD datastructure to represent DFAs as they can become memory extensive. On the other hand, on these benchmarks, Lisa and Lisa-Explicit benefit from the shallowest unrolling. Lisa2-Spot unrolls the formula deeper than Lisa and Lisa-Explicit, resulting in many more composition steps. The runtime of Lisa2-Spot compared to Lisa-Explicit is further affected as the underlying datastructure of Spot's labeled graphs and ROBDDs are known to result in slower compositions.

A closer examination of this class of benchmark revealed a potential avenue for improvement. While the formulas did not have duplicates, they had several *symmetric subformulas* upto propositional isomorphism. Further optimizations based on leveraging such similarities within such formulas could further improve the performance of LTLf-to-DFA conversion tools, including ours.

Lisa2-Spot vs. Lisa2-Mona.: We compare the performance of Lisa2-Spot and Lisa2-Mona. Note that here the underlying algorithm is identical. The only difference between the two is the choice of datastrcture for DFA representations. As a result, we expect this experiment to highlight the impact of datastructure on a tool's performance.

Our observations confirm that the datastructure plays a vital role in a tool's performance, as we observe that the tools Lisa2 Mona and Lisa2-Spot display the same differences displayed by the underlying datastrcuture, i.e. the observed trend is that Lisa2-Mona consumes more memory but is faster while Lisa2-Spotmay be slower but it consumes lesser

Cases	Lisa2-Spot	Lisa2-Spot (Dup. Rem. only)	Lisa2-Mona	Lisa2-Mona (Dup. Rem. only)
nim_01_01	**10**	20	10	10
nim_01_02	20	20	30	**20**
nim_01_03	40	40	**30**	40
nim_01_04	**50**	60	50	50
nim_01_05	80	**70**	**50**	70
nim_01_06	100	100	**80**	90
nim_01_07	**110**	120	**90**	110
nim_01_08	160	**150**	**120**	140
nim_01_09	200	**170**	**190**	230
nim_01_10	240	**210**	230	**210**
nim_01_11	330	**290**	380	**280**
nim_01_12	360	**350**	**420**	440
nim_01_13	560	**500**	**620**	640
nim_01_14	780	**570**	**820**	870
nim_01_15	910	**690**	2220	**1750**
nim_01_16	810	810	9780	**4270**
nim_01_17	810	**780**	**49100**	50760
nim_01_18	1300	**1170**	-	-
nim_01_19	1490	**1400**	108890	**81370**
nim_01_20	1820	**1700**	-	-
nim_02_01	**30**	40	**20**	30
nim_02_02	80	80	40	40
nim_02_03	180	180	**80**	90
nim_02_04	**380**	400	**130**	140
nim_02_05	820	**770**	**200**	220
nim_02_06	2430	**2240**	350	**330**
nim_02_07	8890	**4870**	**480**	510
nim_02_08	76890	**66990**	1150	1150
nim_02_09	348840	**276800**	2540	**2380**
nim_02_10	**466940**	4576480	**3950**	5000
nim_02_11	-	2953960	-	6930
nim_02_12	452700	**299940**	**17690**	18400
nim_02_13	-	13147500	-	-
nim_02_14	-	-	-	-
nim_02_15	**3970670**	4717260	-	-
nim_03_01	110	**100**	60	60
nim_03_02	810	**700**	**160**	170
nim_03_03	27080	**8580**	1110	**640**
nim_03_04	**91610**	173470	2110	**1800**
nim_03_05	396780	**314470**	**4000**	5810
nim_03_06	**1215350**	9473690	7270	-
nim_04_01	1050	**890**	**180**	210
nim_04_02	**75430**	104110	2540	**2120**
nim_04_03	1670340	**1093050**	-	-
nim_04_04	**6634310**	14091500	-	-
nim_05_01	23500	**17230**	1310	**1070**
nim_05_02	**1718010**	5659390	-	-

TABLE III: Ablation Study: Runtime in millisecs for nim benchmarks on Lisa2 and its version with the duplicate removal optimization (i.e. no semantic transformation) only. The **lower runtime** is in bold. Timeout 4hrs.

memory, hence is capable to solve more benchmarks.

These observations further highlight the need for fair comparisons in LTLf-to-DFA conversions that we raised earlier, hence reflects on the importance of tools supporting both datastrucutres for DFA representation.

C. Ablation Study

Finally, we perform an ablation study to examine the effect of each optimization individually. Together the optimizations of duplicate removal and syntactic transformation reduce the number of composition operations. We are interested in studying their effects individually. For this, we compare the performance of Lisa2 (under each DFA datastructure choice) against its own version in which the syntactic transformation has been disabled, i.e., they only applied duplicate removal.

Figure 5 demonstrates the performance of Lisa2-Spot and Lisa2-Mona against their variants that perform duplicate removal only. Apriori, one would imagine that compounding reduction in composition steps through duplicate removal and syntactic transformation would result in improved performance (both in number of benchmarks solved and runtime). However, Figure 5 demonstrates that in some cases (nim_02) the variant that only performed duplicate removal solved more benchmarks. This surprising result led us to further examine the runtime of these tools, shown in Table III. This reveals that there are a significant number of cases where only performing duplicate removal performed better than compounding both optimizations and there are equally many cases where compounding optimizations displayed the stronger runtime performance. In either case, the overall runtime performance is still an improvement over prior state-of-the-art tools.

In order to understand this behavior, we first ascertained that each optimization consumes such a negligible amount of time that it cannot be considered to be the reason behind runtime decline. Similarly, we ascertained that each optimization contributed to reducing the number of composition steps.

We conclude that the unpredictability, despite the reduction in number of composition steps, arises due to the creation of new nodes (new subformulas) after syntactic transformation. To elaborate further, syntactic transformations may result in creating subformulas that were not originally present in the input formula. It is possible that the new formulas are such that even though their DFA construction may require fewer composition steps, these steps may be more expensive as an intermediate DFA may be difficult to create. This results in unpredictability in performance when both optimizations are compounded. In contrast, duplicate removal never creates any new node (new subformula). It only reduces the number of times some nodes may be computed. Hence, duplicate removal will always reduce the overall runtime.

VIII. Concluding Remarks

This work presents Lisa2 which incorporates a series of simple-yet-effective optimizations for compositional approaches for LTLf-to-DFA conversion. Empirical evaluations of this tool displays significant performance improvement, especially on structured benchmarks derived from real-world scenarios: Lisa2 solves $\sim$50% more benchmarks and shows runtime improvement in the range of 1.5x-8000x.

Our optimizations are based on reducing the number of composition steps required to construct the desired DFA. Despite the remarkable performance of Lisa2, our experiments reveal that simply reducing the number of composition steps may not be sufficient, especially if the reduction is accompanied with the creation of new subformulas for which DFA construction may be hard to generate.

We also emphasize on the need for fair comparison to compare algorithmic advances. This is crucial for LTLf-to-

DFA conversion as the choice of datastructure for DFAs have a significant impact on a tool's performance.

Acknowledgements: We thank Marco Favorito and Kuldeep Meel for their help in setting up baseline tools. We thank the anonymous reviewers for their valuable feedback. This work has been supported by the EPSRC through grant EP/X021513/1 and Georgia Institute of Technology's Presidential Undergraduate Research Award for Fall 2023.

REFERENCES

[1] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI*. AAAI Press, 2013, pp. 854–860.

[2] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE, 1977, pp. 46–57.

[3] A. Camacho, E. Triantafillou, C. Muise, J. Baier, and S. McIlraith, "Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, no. 1, 2017.

[4] G. De Giacomo, F. M. Maggi, A. Marrella, and F. Patrizi, "On the disruptive effectiveness of automated planning for ltlf-based trace alignment," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[5] J. A. Baier and S. McIlraith, "Planning with temporally extended goals using heuristic search," in *ICAPS*. AAAI Press, 2006, pp. 342–345.

[6] M. Lahijanian, S. Almagor, D. Fried, L. E. Kavraki, and M. Y. Vardi, "This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction." in *AAAI*. AAAI Press, 2015, pp. 3664–3671.

[7] R. Brafman, G. De Giacomo, and F. Patrizi, "LTLf/LDLf non-markovian rewards," in *AAAI*, vol. 32, no. 1, 2018.

[8] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith, "LTL and beyond: Formal languages for reward function specification in reinforcement learning." in *IJCAI*, vol. 19, 2019, pp. 6065–6073.

[9] K. Jothimurugan, S. Bansal, O. Bastani, and R. Alur, "Compositional reinforcement learning from logical specifications," *Advances in Neural Information Processing Systems*, vol. 34, pp. 10 026–10 039, 2021.

[10] M. Pesic, D. Bosnacki, and W. M. P. van der Aalst, "Enacting declarative languages using LTL: avoiding errors and improving performance," in *SPIN*. Springer, 2010, pp. 146–161.

[11] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, and A. Wells, "Model checking strategies from synthesis over finite traces," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2023, pp. 227–247.

[12] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," in *CAV*. Springer, 1999, pp. 172–183.

[13] S. Bansal, Y. Li, L. Tabajara, and M. Vardi, "Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications," in *AAAI*, vol. 34, no. 06, 2020, pp. 9766–9774.

[14] G. De Giacomo and M. Favorito, "Compositional approach to translate LTLf/LDLf into deterministic finite automata," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 31, 2021, pp. 122–130.

[15] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, "From spot 2.0 to spot 2.10: What's new?" in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 174–187.

[16] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, "Mona: Monadic second-order logic in practice," in *TACAS*. Springer, 1995, pp. 89–110.

[17] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[18] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*. Springer, 1995, pp. 3–18.

[19] N. Klarlund and A. Møller, *MONA Version 1.4: User Manual*, Jan 2001.

[20] L. M. Tabajara and M. Y. Vardi, "Partitioning techniques in LTLf synthesis," in *IJCAI*. AAAI Press, 2019, pp. 5599–5606.