Hybrid Compositional Reasoning for Reactive Synthesis from Finite-Horizon Specifications*

Suguman Bansal,[1] Yong Li,[2,3] Lucas M. Tabajara,[1] Moshe Y. Vardi[1]
[1]Department of Computer Science, Rice University
[2]State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Science
[3]University of Chinese Academy of Sciences
{suguman, lucasmt}@rice.edu, liyong@ios.ac.cn, vardi@cs.rice.edu

Abstract

LTLf synthesis is the automated construction of a reactive system from a high-level description, expressed in LTLf, of its finite-horizon behavior. So far, the conversion of LTLf formulas to deterministic finite-state automata (DFAs) has been identified as the primary bottleneck to the scalabity of synthesis. Recent investigations have also shown that the size of the DFA state space plays a critical role in synthesis as well.

Therefore, effective resolution of the bottleneck for synthesis requires the conversion to be time and memory performant, and prevent state-space explosion. Current conversion approaches, however, which are based either on explicit-state representation or symbolic-state representation, fail to address these necessities adequately at scale: Explicit-state approaches generate minimal DFA but are slow due to expensive DFA minimization. Symbolic-state representations can be succinct, but due to the lack of DFA minimization they generate such large state spaces that even their symbolic representations cannot compensate for the blow-up.

This work proposes a *hybrid* representation approach for the conversion. Our approach utilizes both explicit and symbolic representations of the state-space, and effectively leverages their complementary strengths. In doing so, we offer an LTLf to DFA conversion technique that addresses all three necessities, hence resolving the bottleneck. A comprehensive empirical evaluation on conversion and synthesis benchmarks supports the merits of our hybrid approach.

1 Introduction

Reactive synthesis is the automated construction, from a high-level description of its desired behavior, of a reactive system that continuously interacts with an uncontrollable external environment (Church 1957). This declarative paradigm holds the promise of simplifying the task of designing provably correct reactive systems.

This work looks into the development of reactive synthesis from specifications in Linear Temporal Logic over finite traces (LTLf), or *LTLf synthesis*, for short. LTLf is a specification language that expresses rich and complex temporal behaviors over a *finite* time horizon (Baier and McIlraith 2006; De Giacomo and Vardi 2013). This formalism has found application in specifying task plans in robotics (He et al. 2017; Lahijanian et al. 2015), safety-critical objectives (Zhu et al. 2017a), business processes (Pesic, Bosnacki, and van der Aalst 2010), and the like.

Seminal results have established that LTLf synthesis is 2EXPTIME-complete (De Giacomo and Vardi 2015). Since then, several undertakings have led to algorithmic solutions for synthesis (De Giacomo and Vardi 2015; Camacho et al. 2018). The current state-of-the-art reduces synthesis to a reachability game played on a deterministic finite-state automaton, or DFA (Zhu et al. 2017b). The DFA is obtained by converting the input LTLf specification into a DFA that recognizes the same language. This conversion has been identified as a primary scalability bottleneck in synthesis (Zhu et al. 2017b). This is not surprising as the DFA is known to be double-exponential in the size of the specification in the worst case (Kupferman and Vardi 1999). In order to be effective for synthesis the conversion must, in addition to being time and memory performant, also prevent state-space explosion, as recent investigations have discovered that the efficiency of solving the game on a DFA is strongly affected by the size of the state space (Tabajara and Vardi 2019). This work contributes towards the development of LTLf-to-DFA conversion techniques that are aimed at advancing the scalability of LTLf synthesis.

Prior works on LTLf-to-DFA conversion have led to two contrasting algorithmic approaches. In the first approach (Zhu et al. 2017b), the state-space of the DFA is represented explicitly, the construction is syntax driven, and the DFA is aggressively minimized. This approach first converts LTLf to an equivalent first-order-logic formula and then constructs a DFA for this formula using the MONA tool (Henriksen et al. 1995). The MONA algorithm first produces the binary syntax tree of the specification, then traverses the tree bottom-up while constructing the *minimal* DFA at each node. Consequently, it constructs the final DFA at the root of the tree in its canonical minimal form. Aggressive minimization can often prevent state-space explosion, as for many specifications arising from real-life situations the minimal DFAs are rarely more than exponential in the size of the specification, as opposed to double exponential (Tabakov,

*All authors are corresponding authors

Rozier, and Vardi 2012). Yet, an exponential DFA might still be too large if the set of states is represented explicitly, and the overhead caused by aggressive DFA minimization grows rapidly with specification size.

The second approach, inspired by (Tabajara and Vardi 2019), represents the DFA state space symbolically, uses a compositional construction, and avoids minimizing the DFAs. In compositional constructions, the specification is decomposed into multiple smaller sub-specifications for which explicit DFA conversion is tractable. These intermediate DFAs are then composed to get the final DFA. The symbolic representation encodes the state space of a DFA in a logarithmic number of bits, potentially achieving a polynomial representation even for an exponential-sized DFA, depending on the complexity of the DFA's structure. The existing compositional approach takes advantage of this by representing the intermediate DFAs symbolically. In this case, the DFAs are composed by simply taking the symbolic product *without* performing minimization. The problem with this, however, is that each symbolic product results in a DFA with a larger state space than its minimal DFA, as no minimization is performed. When the number of symbolic products is large, the overhead in the size of the state space magnifies. Because of this, this approach ultimately produces a state space that is so enlarged that not even the succinct symbolic representation can compensate for the blow-up.

The key issue with both approaches is that their critical operation is effective at small scale but becomes inhibitory at large scale. Explicit approaches aggressively perform minimization, which is efficient on small DFAs but expensive on larger ones. Meanwhile, symbolic approaches perform symbolic products without minimization. While few symbolic products are manageable, too many products may lead to a large blow-up in the size of the state space.

This work proposes a novel compositional approach that is able to overcome the drawbacks of both existing approaches. Our approach utilizes a *hybrid* state-space representation, i.e., at different times it uses both the explicit and symbolic state representations for the intermediate DFAs. The core idea is to use explicit-state representation for the intermediate DFAs as long as minimization is not prohibitively expensive, and to switch over to symbolic state representation as soon as that occurs. This way, our hybrid-representation approach applies explicit state representation to small DFAs, and also delays the point at which switch-over to symbolic representation occurs, thus ensuring that fewer symbolic products have to be performed to generate the final DFA. Therefore, by finding a balance between the two representations, our hybrid appoach is able to extract their benefits and mitigate their weaknesses.

We have implemented our LTLf-to-DFA conversion algorithm, and its extension to LTLf synthesis via reachability games, in tools called LISA and LISASYNT, respectively. A comprehensive empirical analysis reveals the merits of the proposed hybrid compositional approach on both DFA conversion and LTLf synthesis, as each tool outperforms the current state-of-the-art in runtime and memory consumption. In addition, the DFAs generated from LISA have size comparable to the minimal DFA and significantly smaller than those obtained from pure symbolic-state methods.

2 Preliminaries

Linear Temporal Logic over Finite Traces

Linear Temporal Logic over finite traces (LTLf) (Baier and McIlraith 2006; De Giacomo and Vardi 2013) extends propositional logic with finite-horizon temporal operators. In effect, LTLf is a variant of LTL (Pnueli 1977) that is interpreted over a finite rather than infinite trace. The syntax of an LTLf formula over a finite set of propositions Prop is identical to LTL, and defined as $\varphi := a \in \text{Prop} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \text{X}\varphi \mid \varphi \,\text{U}\, \varphi \mid \text{F}\varphi \mid \text{G}\varphi$. Here X (Next), U (Until), F (Eventually), G (Always) are temporal operators. The semantics of LTLf can be found in (De Giacomo and Vardi 2013). W.l.o.g., we assume that every LTLf formula φ is written as a conjunction of LTLf subformulas i.e. $\varphi = \bigwedge_{i=1}^{n} \varphi_i$. The language of an LTLf formula φ, denoted by $\mathcal{L}(\varphi)$, is the set of finite words over 2^{Prop} that satisfy φ.

LTLf synthesis is formally defined as follows:

Definition 1 (LTLf Synthesis)**.** *Let φ be an LTLf formula over Prop $= \mathcal{I} \cup \mathcal{O}$ where the set of input variables $\mathcal{I}$ and output variables $\mathcal{O}$ are two disjoint sets of propositions. We say φ is realizable if there exists a strategy $\gamma : (2^{\mathcal{I}})^+ \to 2^{\mathcal{O}}$ such that for every infinite sequence $\lambda = I_0, I_1, \cdots \in (2^{\mathcal{I}})^\omega$ of interpretations over $\mathcal{I}$, there exists $m \geq 0$ such that $\rho = (I_0 \cup \gamma(I_0)), (I_1 \cup \gamma(I_0, I_1)), \cdots, (I_m \cup \gamma(I_0, \cdots, I_m))$ satisfies φ. The problem of LTLf synthesis is to decide whether a given φ is realizable and to construct such a strategy if so.*

Intuitively, LTLf synthesis can be perceived as a game between an external environment and the desired system that take turns to assign values to input and output propositions, respectively. The system responds to the environment inputs using the strategy γ. The game is won by the system if its strategy is able to guarantee that the resultant input-output sequence will satisfy formula φ after a finite number of turns. In our formulation of LTLf synthesis, like in (Tabajara and Vardi 2019), the environment plays first. Alternatively, the system may play first (Zhu et al. 2017b). Solving the alternative formulation requires only slight changes to the algorithm presented in (§ 5). We adhere to the formulation in Definition 1 in this paper as our benchmarks assume that formulation and all tools being compared support it.

DFA and Its Representations

A *deterministic finite automaton* (DFA) (Thomas, Wilke, and others 2002) is a tuple $D = (\Sigma, S, \iota, \Delta, F)$ where Σ is a finite set of symbols (called an alphabet), S is a finite set of states, $\iota \in S$ is the initial state, $F \subseteq S$ is the set of accepting states and $\Delta : S \times \Sigma \to S$ is the transition function. A finite word $w = w_0 \dots w_n \in \Sigma^*$ has a *run* $\rho = s_0 \dots s_{n+1} \in S^+$ in D if for all $i \in \{0, \dots n\}$ we have that $s_{i+1} = \Delta(s_i, w_i)$ and $s_0 = \iota$. A run $\rho = s_0 \dots s_{n+1}$ is an *accepting run* in D if $s_{n+1} \in F$. A word w is in the language of D, $\mathcal{L}(D)$, if w has an accepting run in D. A DFA is said to be *minimal* if the language represented by that DFA cannot be represented by another DFA with fewer states.

Every LTLf formula φ over Prop can be converted into a DFA D with alphabet $\Sigma = 2^{\text{Prop}}$ (De Giacomo and Vardi

2013) such that $\mathcal{L}(D) = \mathcal{L}(\varphi)$. If this DFA is constructed in a form that explicitly enumerates all DFA states, we call it an *explicit-state* representation. A DFA over the alphabet $\Sigma = 2^{\text{Prop}}$ can also be compactly represented *symbolically*, by also encoding the state space using a logarithmic number of propositions. The *symbolic-state* representation of a DFA $D = (2^{\text{Prop}}, S, \iota, \Delta, F)$ is a tuple $\mathcal{D} = (\mathcal{S}(\mathcal{Z}), \mathcal{T}(\mathcal{Z}, \text{Prop}, \mathcal{Z}'), \mathcal{F}(\mathcal{Z}))$. In this representation, $\mathcal{Z} = \{z_1, \dots z_n\}$ are propositions encoding the state space S, with $n = \lceil \log |S| \rceil$, and their primed counterparts $\mathcal{Z}' = \{z'_1, \dots z'_n\}$ encode the next state. Each state $s \in S$ corresponds to an interpretation $Z \in 2^{\mathcal{Z}}$ over propositions $\mathcal{Z}$. When representing the next state of the transition function, the same encoding is used for an interpretation Z' over $\mathcal{Z}'$. Then, $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{F}$ are Boolean formulas representing ι, Δ and F, respectively. $\mathcal{S}(\mathcal{Z})$ is satisfied only by the interpretation of the initial state ι over $\mathcal{Z}$. $\mathcal{T}(\mathcal{Z}, \text{Prop}, \mathcal{Z}')$ is satisfied by interpretations $Z \in 2^{\mathcal{Z}}$, $P \in 2^{\text{Prop}}$ and $Z' \in 2^{\mathcal{Z}'}$ iff $\Delta(s, P) = s'$, where s and s' are the states corresponding to Z and Z'. Lastly, $\mathcal{F}(\mathcal{Z})$ is satisfied by the interpretation Z over $\mathcal{Z}$ corresponding to state $s \in S$ iff $s \in F$. The intersection of two DFAs $D_1 = (\mathcal{S}_1, \mathcal{T}_1, \mathcal{F}_1)$ and $D_2 = (\mathcal{S}_2, \mathcal{T}_2, \mathcal{F}_2)$, denoted $D_1 \wedge D_2$, is given by $(\mathcal{S}_1 \wedge \mathcal{S}_2, \mathcal{T}_1 \wedge \mathcal{T}_2, \mathcal{F}_1 \wedge \mathcal{F}_2)$. In this paper, all Boolean formulas, including $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{F}$ of a symbolic DFA, will be encoded using *Reduced Ordered Binary Decision Diagrams* (BDDs) (Bryant 1986).

DFA Game

A *DFA game* is a reachability game between two players, called the *environment* and the *system*, played over a DFA with alphabet $2^{\mathcal{I} \cup \mathcal{O}}$. The environment player assigns values to the input variables $\mathcal{I}$, while the system assigns values to the output variables $\mathcal{O}$. The DFA game starts at the initial state of the DFA. At each round of the game, first the environment chooses an assignment I to the $\mathcal{I}$ variables, and then the system will choose an assignment O to the $\mathcal{O}$ variables. The combined assignment $I \cup O$ determines the unique state the game moves to according to the transition function of the DFA. The system *wins* the game if the game reaches an accepting state of the DFA. *Solving a DFA game* corresponds to determining whether there exists a strategy for the system to always win the game.

DFA games are known to be solvable in polynomial time with respect to the number of states (Mazala 2002). The algorithm determines if the initial state is a *winning state*, i.e., a state that is either accepting or from which, for every assignment I to the $\mathcal{I}$ variables, the system can always choose an assignment O to the $\mathcal{O}$ variables that leads to a winning state. More details will be given in (§ 5). If the initial state is a winning state, then there exists a winning strategy that can be represented by a Mealy machine that determines the output of the system given the current state and input. For more details, refer to (Tabajara and Vardi 2019).

3 Related Work

LTLf to DFA Conversion There are two commonly used approaches for the conversion currently. In the current state-of-the art approach, the LTLf formula is translated into first-order logic over finite traces, and then converted into a DFA by MONA, a more general conversion tool from monadic second-order logic to DFA (Henriksen et al. 1995). The first LTLf synthesis tool SYFT utilizes this method for DFA generation (Zhu et al. 2017b).

An alternative approach, used by the tool SPOT (Duret-Lutz et al. 2016), is to translate the LTLf formula into an LTL formula with equivalent semantics, convert this formula into a Büchi automaton (Gerth et al. 1995), and then transform this Büchi automaton into a DFA. Both approaches generate a DFA in explicit state-space representation.

DFA vs. NFA NFAs are more general than DFAs. In fact, NFAs can be constructed from an LTLf formula in a single-exponential blow-up as opposed to the double-exponential blow-up incurred for DFA construction. Various approaches for LTLf-to-NFA with single-exponential blow-up have been described such as (Baier and McIlraith 2006; De Giacomo and Vardi 2015). Yet, in practice, single exponential NFA conversion tools do not perform as well as DFA conversion tools. (Tabakov, Rozier, and Vardi 2012) shows that minimal DFAs from LTLf formulas tend to be orders of magnitude smaller than their NFA counterparts constructed from implementations of the single-exponential algorithms.

LTLf Synthesis As aforementioned, current state-of-the-art tool SYFT (Zhu et al. 2017b) uses MONA to construct an explicit-state DFA, then converts this DFA into a symbolic representation in order to solve the game using a symbolic fixed-point computation. The explicit-state DFA construction has been identified as the primary bottleneck to SYFT as the length of the formula increases. Therefore, recent attempts in synthesis have been made to avoid the explicit DFA construction. We describe these attempts below.

A recent approach attempted to avoid the full construction by instead decomposing the specification into conjuncts, then converting each conjunct to an individual DFA (Tabajara and Vardi 2019). Since these conjuncts are smaller formulas, their explicit-state DFAs can be constructed efficiently. The smaller DFAs are then converted into a symbolic representation and the game is solved over this *decomposed symbolic representation*. While the construction was indeed more efficient in terms of time and memory, the resulting DFA had a much larger state space. This severely decreased the performance of the game-solving algorithm, rendering a poorly scaling procedure for LTLf synthesis.

In another attempt to avoid explicit DFA construction, (Camacho et al. 2018) first constructs an NFA from the formula and then reduces synthesis to fully-observable nondeterministic (FOND) planning. The NFA is determinized on-the-fly during the planning phase. Even here, the specification is decomposed into conjuncts, which are separately converted to NFAs and used to encode to FOND. Despite the generalization to NFAs, in practice FOND-based methods rely on DFA conversion tools since they are more competitive than existing NFA construction tools that incur a single-exponential blow up. Previous experiments suggest the FOND-based approach is complementary with the approach based on explicit DFA construction, each being able

to solve instances that the other cannot.

Compositional Techniques in Temporal Synthesis Both (Tabajara and Vardi 2019) and (Camacho et al. 2018) benefit from compositional techniques as they both decompose the input formula into conjuncts before construction of the respective automata. Application-specific decomposition has also been shown to lead to an orders-of-magnitude improvement in LTLf synthesis for robotics (He et al. 2019).

A precedent for compositional techniques exists also in synthesis of LTL over infinite traces. Some state-of-the-art tools are STRIX (Meyer, Sickert, and Luttenberger 2018) and ACACIA+ (Bohy et al. 2012). STRIX decomposes the formula semantically, i.e., it generates a subformula if it belongs to a restricted fragment of LTL such as safety LTL or co-safety LTL. This way it benefits from constructing automaton using more efficient fragment-specific algorithms. On the other hand, ACACIA+ decomposes the formula into conjuncts, which are each solved as a separate safety game. The final solution is obtained by composing solutions from the separate safety games.

4 Hybrid Compositional DFA Generation

This section describes the primary contribution of this work. We present a novel compositional approach for LTLf-to-DFA conversion. Our approach is based on using a hybrid-state representation, i.e., at different times it uses both explicit and symbolic-state representations for intermediate DFAs, as opposed to prior works in which only one of the two state-representations is used (Zhu et al. 2017b; Camacho et al. 2018; Tabajara and Vardi 2019). By diligent application of both representations, our hybrid approach is able to leverage their complementary strengths and render an algorithm that is not only competitive time- and memory-wise, but also generates DFAs with small number of states.

Our compositional approach is comprised of two phases, called the *decomposition phase* and the *composition phase*. In the decomposition phase, the input formula is first *decomposed* into smaller subformulas which are then converted into their equivalent DFAs using standard algorithms. In the composition phase, the intermediate DFAs are *composed* to produce the final DFA. We describe each phase for our hybrid approach in detail below. The formal description has been deferred to the Appendix in (Bansal et al. 2019).

Decomposition Phase

The decomposition phase is the first step in our algorithm. This phase receives the LTLf formula φ as input. We make an assumption that the formula is given as the conjunction of multiple small LTLf subformulas, i.e., $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ where each φ_i is an LTLf formula in itself. This assumption has been adopted as a standard practice in synthesis domains as large specifications arising from applications tend to exhibit this form (Filiot, Jin, and Raskin 2010; 2011).

We interpret formula φ as an n-ary syntax tree as opposed to a binary-tree. Consequently, the input formula $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ is decomposed into n-subformulas $\varphi_1, \ldots, \varphi_n$. Then each of these subformulas φ_i is converted into its minimal DFA D_i in explicit-state representation. This can be performed by an existing tool (De Giacomo and Vardi 2013; Duret-Lutz et al. 2016; Henriksen et al. 1995; Kupferman and Vardi 1999). More advanced decomposition schemes could be adopted from (Camacho et al. 2018).

The rationale behind this step is that existing explicit-state tools are efficient in generating minimal DFA for small formulas. Since the subformulas are typically small in length, we are able to benefit from existing literature in this step.

Composition Phase

The composition phase receives the minimal DFAs D_i for subformulas φ_i in the previous phase, which are represented with explicit states. Our goal in this phase is to construct a DFA corresponding to φ. In theory, this can be obtained by simply taking the intersection of DFAs D_i. In practice, the intersection of n DFAs may lead to state-space explosion since DFA intersection is done by performing their product construction. Therefore, the main focus of the composition phase is about how to efficiently construct the intersection without incurring state explosion. We discuss the salient features of our algorithm before describing it in detail.

Briefly speaking, we perform the composition of DFAs in iterations. In each iteration, two DFAs are selected based on a *dynamic smallest-first heuristic*, which will be described below, and removed from the set. A new DFA is formed by the product of the two selected DFAs. The new DFA will be minimized based on a *selective DFA heuristic*, which is also described below. The new DFA is then inserted back into the set. The new set is the input to the next iteration. This continues until only one DFA remains, which is presented as the final DFA. In the following, we denote by S_j the set of DFAs at the j-th iteration. Then $S_1 = \{D_1, \ldots, D_n\}$, and $S_n = \{D\}$ where D is the final output DFA.

In contrast to prior works which either use explicit states or symbolic states, the central feature of our algorithm is that it uses hybrid representation for DFAs, i.e., in different iterations all DFAs in S_j are either represented in explicit- or symbolic-state form. Initially, all DFAs in S_1 are in explicit-state form. This continues while the DFAs in S_j have a small number of states, since the product and minimization of DFAs are efficient for small DFAs with explicit-state representation. But as some DFAs in S_j grow in size they require more memory and longer time to perform minimization. So, as soon as some DFA in S_j reaches a large number of states, all DFAs in S_j are converted into symbolic-state representation, in which the DFAs are represented more succinctly. By this time, hopefully, we are left with few DFAs in the set S_j. Here onwards, all DFAs are represented in symbolic form until the end of the algorithm. Therefore, fewer DFAs in S_j implies fewer symbolic products need to be performed, and hence limits the blow-up in state-space of the final DFA. This way, our algorithm balances the strengths of both approaches, mitigates their individual drawbacks, and efficiently generates a small DFA, if not the minimal.

We now describe the two heuristics, namely *dynamic smallest-first composition of DFAs* and *selective DFA minimization* abbreviated to DSF and SDM, respectively.

We first discuss DSF, which is used to decide which two DFAs should be composed in each iteration. We observe

that the order in which intersection of DFAs is performed does not affect the correctness of the final DFA since both Boolean conjunction and DFA intersection are associative and commutative operations. In theory, we can design any criteria to select two DFAs to be composed at each iteration. In practice, a careless choice of the two DFAs may produce an unnecessarily large intermediate DFA that causes the algorithm to fail at the composition phase due to the large memory footprint. Therefore, we aim to find an order that can optimize time and space in the composition phase. To help with that we use DSF, which as the name suggests chooses the smallest two DFAs in each iteration. The DFAs with explicit states are chosen based on the number of states, while the DFAs with symbolic-state representation are chosen based on the number of nodes in the BDD representation of the transition function. The intuition behind this heuristic is that if the algorithm would fail on the composition of the smallest two DFAs in that iteration, then it would probably fail on the composition of all other pairs of DFAs as well.

Next we discuss SDM, which decides when it is beneficial to perform DFA minimization after the intersection of DFAs in each iteration. DFA minimization has been proved to be critical to the performance of DFA generation in (Henriksen et al. 1995) as it helps in maintaining a smaller number of states, which is also one of our critical parameters. However, it is also an expensive operation. Currently, the best known complexity for minimization are $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2)$ for explicit- and symbolic-state representations, respectively (Hopcroft 1971; Wimmer et al. 2006). Therefore, there is a tension between reducing the number of states and achieving efficiency. To resolve this, we conducted an empirical study to evaluate the effect of minimization. We observed that in most cases, minimization reduces the number of states by 2-3 times. While this is significant when the states are represented explicitly, in symbolic-state representation this leads to a reduction in 1-2 state variables only. Therefore, we adhere to the SDM heuristic in which we minimize intermediate DFAs in explicit-state representation only. There are two advantages to this. First, since minimization is performed on explicit-state representation only, by virtue of our algorithm design this occurs only when the DFAs are small. For these, the time spent in minimization is so low that it is worth maintaining minimal DFAs. Second, by maintaining minimal DFAs in the explicit-form, the algorithm delays the switch over to symbolic form as the DFA sizes take longer to reach the thresholds. This leads to fewer symbolic products, which results in curbing the amount of blow-up in state-space.

A semi-formal description of the steps of the algorithm are given below. The complete description has been deferred to (Bansal et al. 2019).

Step 0. (Initial) We are given input formula $\varphi = \bigwedge_{i=1}^{n} \varphi_i$, and *switch-over threshold values* $t_1, t_2 > 0$. The parameters t_1 and t_2 correspond to the thresholds for the numbers of states in an individual DFA and in the product of two DFAs, respectively, to trigger the symbolic representation.

Step 1. (Decomposition) Construct the minimal DFA D_i in explicit-state representation for all $i \in \{1, \ldots, n\}$. Create the set $S_1 = \{D_1, \ldots, D_n\}$.

Step 2. (Explicit-state Composition) For $j \in \{1, \ldots, n-1\}$, let $S_j = \{M_1, \ldots, M_{n-j+1}\}$ be the set of DFAs in the j-th iteration.

If S_j has only one DFA, return that as the solution.

Otherwise, if the DFAs in S_j become too large, proceed to Step 3. Assume w.l.o.g. that M_1 and M_2 are the two DFAs chosen by the DSF heuristic. Let $|A|$ denote the number of states in a DFA A represented in explicit-state form. If $\min(|M_1|, |M_2|) > t_1$ or $(|M_1| \cdot |M_2|) > t_2$, move to Step 3. Let k be the iteration in which this occurs, i.e. when $j = k$.

Otherwise, as per SDM, construct DFA P by minimization of $M_1 \cap M_2$. Then, create $S_{j+1} = \{P, M_3, \ldots, M_n\}$ for the next iteration, and repeat Step 2.

Step 3. (Change State Representation) Convert all DFAs in $S_k = \{M_1, \ldots, M_{n-k+1}\}$ from explicit-state to symbolic-state representation, and proceed to Step 4. Note that the state space of each DFA M_i is encoded symbolically using a different set of state variables $\mathcal{Z}_i$, where all $\mathcal{Z}_i$ are disjoint. Since no more minimization occurs after this point, the total set of state variables $\mathcal{Z} = \mathcal{Z}_1 \cup \ldots \cup \mathcal{Z}_{n-k+1}$ defines the state space of the final DFA.

Step 4. (Symbolic-state Composition) For $j \in \{k, \ldots, n\}$, let $S_j = \{M_1, \ldots, M_{n-i+1}\}$ be the set of DFAs in the j-th iteration.

If S_j has only one DFA, return that DFA as the solution.

Otherwise, assume w.l.o.g. that M_1 and M_2 are the two DFAs chosen by the DSF heuristic. Construct $P = M_1 \wedge M_2$. Recall that, since M_1 and M_2 are in symbolic form, we do not perform DFA minimization of P. Create $S_{i+1} = \{P, M_3, \ldots, M_n\}$ for the next iteration, and repeat Step 4.

5 LTLf Synthesis

LTLf synthesis can be reduced to solving a DFA game played on the DFA corresponding to the formula φ (De Giacomo and Vardi 2015). As explained in (§ 2), this amounts to computing the set of winning states. If the initial state of the DFA is in this set, then the formula is realizable and a winning strategy can be constructed, otherwise not.

In this section, we describe the winning set computation algorithm on a DFA game when its states are represented symbolically. This is a standard least-fixed point algorithm for reachability games with symbolic state space, and is similar to (Zhu et al. 2017b; Tabajara and Vardi 2019). For sake of completion, we summarize the algorithm here.

Let φ be an LTLf formula over disjoint input and output propositions $\mathcal{I}$ and $\mathcal{O}$, respectively, and $\mathcal{G} = (\mathcal{S}(\mathcal{Z}), \mathcal{T}(\mathcal{Z}, \text{Prop}, \mathcal{Z}'), \mathcal{F}(\mathcal{Z}))$ be a symbolic DFA for φ. The DFA game is played on $\mathcal{G}$. In our case, this DFA is obtained from our hybrid compositional approach (§ 4), which we assume is in symbolic form, since explicit-state outputs can easily be converted to symbolic form.

To compute the winning set of $\mathcal{G}$, we compute the least-fixed point of a Boolean formula $W_i(\mathcal{Z})$ that denotes the set of states from which the system can win in at most i steps of the DFA game. Initially, $W_0(\mathcal{Z})$ is the set $\mathcal{F}(\mathcal{Z})$ of accepting states. At each iteration, the algorithm constructs $W_{i+1}(\mathcal{Z})$

from $W_i(\mathcal{Z})$ by adding those states from which the system is guaranteed to reach $W_i(\mathcal{Z})$ in one step. Formally,

$$W_{i+1}(\mathcal{Z}) = W_i(\mathcal{Z}) \vee (\forall \mathcal{I}.\exists \mathcal{O}, \mathcal{Z}'.\mathcal{T}(\mathcal{Z}, \mathcal{I} \cup \mathcal{O}, \mathcal{Z}') \wedge W_i(\mathcal{Z}'))$$

where $W_i(\mathcal{Z}')$ can be obtained from $W_i(\mathcal{Z})$ by substituting variables $\mathcal{Z}$ with $\mathcal{Z}'$. This continues until no more states can be added to $W_{i+1}(\mathcal{Z})$, i.e., until it encounters the first index i such that $W_{i+1}(\mathcal{Z}) \equiv W_i(\mathcal{Z})$. Since the number of states in the DFA is finite, the algorithm is guaranteed to terminate. The initial state is present in the winning set, say $W_{\text{FP}}(\mathcal{Z})$, if $\mathcal{S}(\mathcal{Z}) \implies W_{\text{FP}}(\mathcal{Z})$ holds. Details on winning-strategy construction has been deferred to (Tabajara and Vardi 2019).

In this work, all Boolean formulas for $\mathcal{G}$ and all $W_{i+1}(\mathcal{Z})$ are represented as BDDs. All boolean operations, quantification and variable substitution are available in standard BDD libraries. Finally, $\equiv$ is a constant time operation in BDDs.

The complexity of solving a DFA game is polynomial in the size of the state space. Therefore, the efficiency of LTLf synthesis is heavily affected by the size of the constructed DFA. Therefore, as our hybrid compositional approach generates small (if not minimal) DFAs, these are suitable for synthesis, as witnessed also by our experimental evaluation.

6 Experimental Evaluation

The goal of the empirical analysis is to examine the performance of our hybrid approach in LTLf-to-DFA generation and LTL synthesis against existing tools and approaches.

Implementation Details

Our hybrid compositional LTLf-to-DFA conversion procedure (§ 4) has been implemented in a tool called LISA. LISA has been extended to LISASYNT to perform LTLf synthesis using the winning strategy computation described in (§ 5).

LISA takes an LTLf formula and switch-over thresholds t_1, t_2 as inputs, and outputs a corresponding DFA with symbolic states. The output may not be minimal. For the same inputs, LISASYNT internally invokes LISA, solves the DFA game given by LISA's output, and returns whether the formula is realizable. If so, it can also return a winning strategy.

LISA and LISASYNT have been written in C++. They employ BUDDY (Cohen et al. 2014) as their BDD library for the symbolic representations and operations on DFAs, and take advantage of dynamic variable ordering for the BDDs.

To generate explicit-state minimal DFAs in the decomposition phase, LISA uses SPOT (Duret-Lutz et al. 2016) and the MONA-based method (Henriksen et al. 1995). It borrows the rich APIs from SPOT to conduct DFA intersection and minimization in the explicit-state composition phase. Per se, SPOT APIs are available for ω-automata (automata over infinite words). In order to use the SPOT API for operations over DFAs, LISA stores intermediate explicit DFAs as *weak deterministic Büchi automata* (wDBA) (Dax, Eisinger, and Klaedtke 2007). Intuitively, if the DFA accepts the language $\mathcal{L}$, then its wDBA accepts the language $\mathcal{L} \cdot (\{\text{ loop}\})^\omega$, where loop is a fresh variable not present in Prop. The wDBA can be constructed from the DFA for $\mathcal{L}$ by making the following changes (a) add a new state sink, (b) for each accepting state in the DFA, add a transition from that state to sink on loop, (c) add a transition from sink to itself on loop, (d) make sink the only accepting state in the wDBA. This automaton accepts a word iff its run visits sink infinitely often. Since wDBA is an ω-automaton, we use SPOT APIs for wDBAs to conduct intersection and minimization, both of which return a wDBA as output, in a similar complexity for those operations in a DFA (Dax, Eisinger, and Klaedtke 2007; Kupferman 2018). Lastly, a wDBA for language $\mathcal{L} \cdot (\{\text{ loop}\})^\omega$ can be easily converted back to a DFA for language $\mathcal{L}$.

# States in the minimal DFA	Number of benchmarks solved		
	Mona-based	Lisa-Explicit	Lisa
$\geq$ 1K	111	123	137
$\geq$ 5K	70	82	96
$\geq$ 10K	48	60	74
$\geq$ 50K	13	23	35
$\geq$ 100K	8	16	26
$\geq$ 250K	1	5	12
$\geq$ 500K	0	2	4
$\geq$ 750K	0	2	2
Size unknown	–	–	21**
Total solved	**307**	**338**	**372**

Table 1: DFA construction. Hardness of benchmarks is measured by the size of minimal DFA. **Note: There are 34 benchmarks that were solved only by LISA. Of these, the size of the minimal DFA of 13 benchmarks were identified using a symbolic DFA minimization algorithm (Wimmer et al. 2006). The 21 cases with unknown size are those that could not be minimized even after 24hrs with 190GB.

Design and Setup for Empirical Evaluation [1]

The evaluation has been designed to compare the performance of LISA and LISASYNT to their respective existing tools and approaches. LTLf-to-DFA conversion tools are compared on runtime, number of benchmarks solved, hardness of benchmarks solved (size of minimal DFA) and the number of state variables in the output DFA. LTLf synthesis tools are compared on runtime and the number of benchmarks solved. We conduct our experiments on a benchmark suite curated from prior works, spanning classes of realistic and synthetic benchmarks. In total, we have 454 benchmarks split into four classes: random conjunctions (400 cases) (Zhu et al. 2017b), single counters (20 cases), double counters (10 cases) and Nim games (24 cases) (Tabajara and Vardi 2019). We defer more details to (Bansal et al. 2019).

A good balance between explicit- and symbolic-representation of states is crucial to the performance of LISA, i.e., it is crucial to carefully choose values of the switch-over thresholds t_1 and t_2. Recall the switch is triggered if either the smallest minimal DFA has more than t_1 states, or if the product of the number of states in the two smallest minimal DFAs is more than t_2. Intuitively, we want t_1 to be large enough that the switch is not triggered too

[1] Figures are best viewed online in color.

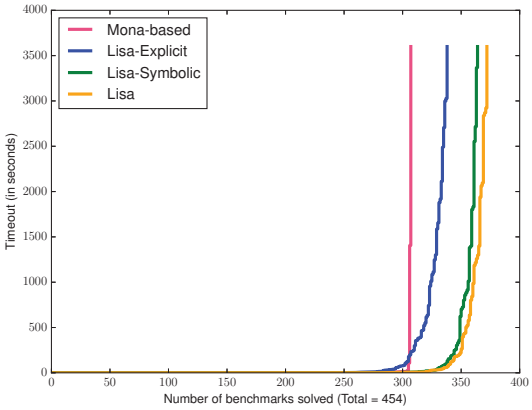

Figure 1: DFA construction. Cactus plot indicating number of benchmarks each tool can solve for a given timeout.

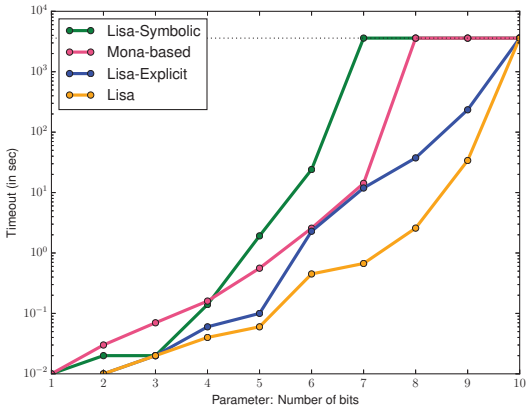

Figure 2: DFA construction. Runtime for double-counter benchmarks. Plots touching black line means time/memout.

soon but small enough that conversion of all DFAs from explicit- to symbolic-state representation is not too expensive. Threshold t_2 is closely related to how effective minimization is, and hence depends on the benchmark class. If the benchmark class is such that minimization reduces the DFA size by only 2-3 times, then we would set t_2 to be a low value. But if the class is such that minimization reduces DFA size by orders of magnitude, as it does for the Nim game class, we set t_2 to a higher value to take advantage of minimization. Currently, these are determined empirically. We set $t_1 = 800$ and $t_2 = 300000$ for the Nim-game class and to $t_1 = 800$ and $t_2 = 2500$ for all other classes.

For experiments on LTLf-to-DFA conversion, we compare LISA to the current-state-of-the-art MONA-based method (Zhu et al. 2017b; Camacho et al. 2018) and two other derivations of LISA. Recall the MONA-based method is a syntax-driven, explicit-state based approach that returns minimal DFAs. The first derivation is LISA-EXPLICIT which is adapted from LISA by setting $t_1 = t_2 = \infty$. Therefore, it is a purely explicit-state compositional approach. Like the MONA-based method, it also generates the minimal DFA, but unlike the former it uses the smallest-first heuristic. The second derivation is LISA-SYMBOLIC, adapted from LISA by setting $t_1 = t_2 = 0$. This corresponds to the compositional, symbolic-state approach referred to in (§ 1).

For experiments on LTLf synthesis, we compared LISASYNT to an enhanced version of SYFT (a tool that uses the MONA-based method for DFA conversion) (Zhu et al. 2017b) that we call SYFT+, SYNKIT (Camacho et al. 2018), and the partitioned approach from (Tabajara and Vardi 2019), referred to as PART. SYFT+ was created by enabling dynamic variable ordering in SYFT. This was necessary for a fair comparison as SYFT, unlike LISASYNT and PART, uses static variable ordering. We observed that SYFT+ shows upto 75% reduction in runtime compared to SYFT. For SYNKIT, we report the results without automata decomposition, since decomposing the formulas in the same granularity used for LISA decreased performance, and we lack methods

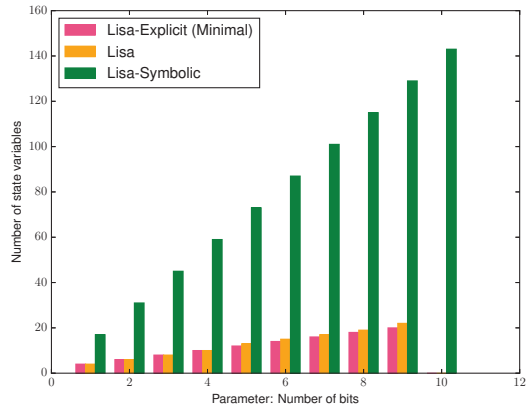

Figure 3: DFA construction. Number of variables needed to symbolically represent the DFA's state-space for double-counter benchmarks. No bar indicates time/memout.

for choosing a better granularity for this tool. This behavior is consistent with observations in (Camacho et al. 2018) and (Tabajara and Vardi 2019) that too much decomposition is detrimental for performance as it leads to state-space explosion. Note that PART uses the same symbolic-state approach as LISA-SYMBOLIC for constructing the DFAs, except that it skips the composition step, instead performing synthesis directly over the initial set of symbolic DFAs S_1. Ultimately, it still suffers from the state-space explosion, only in this case it happens during the winning-state computation.

All experiments were conducted on a single node of a high-performance cluster. Each node consists of four quad-core Intel-Xeon processor running at 2.6 GHz. LTLf-to-DFA conversion experiments were run for 1 hour with 8 GB each, LTLf-synthesis experiments for 8 hours with 32 GB each.

Observations

LISA and LISA-EXPLICIT scale better to larger benchmarks than the MONA-based method, not just solving more total benchmarks but also being able to handle instances of larger scale (Table 1). Compared to MONA, LISA-EXPLICIT is more consistent in solving benchmarks with large minimal DFAs due to the DSF heuristic that enables low memory consumption in intermediate stages. Finally, LISA solves benchmarks with even larger minimal DFAs as it is designed to combine minimal DFAs of explicit state- and succinctness of symbolic-state representation to solve larger formulas.

LISA is the most efficient tool among all four options. This is clear from the cactus plot in Fig. 1. The plot may seem to indicate that LISA only has a slight advantage over LISA-SYMBOLIC. But, on closer inspection we observe that LISA-SYMBOLIC solves most random benchmarks but fares poorly on the realistic ones (see Fig 2). This is because they have more sub-specifications, resulting in a large number of symbolic products. The MONA-based method is still the fastest in generating small DFAs (fewer than 50K states) but memouts soon due to explicit-state representation of DFAs. Finally, LISA-EXPLICIT is a close second but does not scale as well as LISA due to minimization on very large DFAs. LISA has been designed to overcome these deficiencies, and is supported by the current empirical evaluation as well.

LISA mitigates state-space explosion. Even though LISA may not generate the minimal DFAs, we observe that in most cases the state-space of the final DFA produced by LISA is one or two variables more than that of the minimal DFA. This is significantly lower than the number of state variables used by LISA-SYMBOLIC (Fig. 3). Note that LISA-SYMBOLIC fails to solve the double counter benchmarks for $i \geq 7$ (Fig 2). Yet we know the number of state variables immediately after Step 3 (§ 4). Analyzing the benchmarks, we observed that they were split into 3-200 sub-formulas, yet only 1-3 symbolic products were conducted to construct the DFA. This demonstrates that our threshold-values are able to delay the switch-over to symbolic representations and reduce blow-up by the product. This is why the DFAs generated by LISA have comparable sizes to the minimal DFAs. An important future work, therefore, is to design mechanisms to determine the switch-over thresholds at runtime as opposed to relying on user-expertise to assign threshold values.

LISA's small DFAs improve synthesis performance. We evaluate for synthesis on non-random benchmarks only, i.e., sequential counters and Nim games. We chose to disregard random benchmarks as their winning set computation time is negligible, as in those benchmarks the fixed-point is reached in 2-3 iteration irrespective of the DFA size. Figure 4-5 show that LISASYNT solves most benchmarks and is the most efficient tool. We observed that SYFT+ fails because MONA memouts early, while SYNKIT memouts during the planning stage and PART suffers from state-space explosion while solving the game. LISASYNT is resilient to both as LISA consumes low memory by virtue of symbolic representation and small state space.

The time consumed inside the winning set computation during synthesis depends on the number of iterations before the fixed-point is reached. Yet, so far not much focus has been given to optimizing this step as the DFAs generated so far have not been large enough for the number of iterations to become an issue. With LISA's ability to construct large DFAs, we were able to observe that the single and double counter benchmarks can spend more than 90% of the time in the winning set computation, as the number of iterations is exponential in the number of bits (see (Bansal et al. 2019)). This provides concrete evidence of the importance of investigating the development of faster algorithms for winning set computation to improve game-based synthesis.

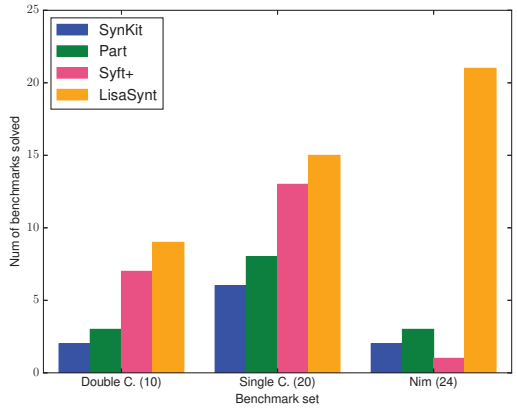

Figure 4: Synthesis. Number of benchmarks synthesized from each non-random benchmark class.

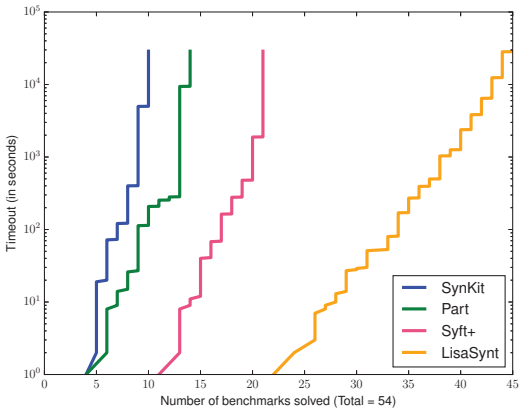

Figure 5: Synthesis. Cactus plot (non-random benchmarks).

7 Concluding Remarks

This work tackles the primary bottleneck in LTLf synthesis-LTLf to DFA conversion. The central problem addressed in this work is the efficient and scalable construction of DFAs with small state space from LTLf specifications, as a step to LTLf synthesis. To the best of our knowledge, ours is the *first* hybrid approach for DFA construction. Our approach combines explicit- and symbolic-state representa-

tions in a manner that effectively leverages their strengths and alleviates their individual shortcomings. Our empirical evaluations on DFA conversion and LTLf synthesis on LISA and LISASYNT outperform the current states of the art, and demonstrate the merit of our hybrid approach. This indicates promise to further develop and explore hybrid approaches for automaton generation for other specification languages as well, and encourages similar investigations into the other building blocks in synthesis algorithms.

Acknowledgement

We thank A. Camacho, A. M. Wells and S. Zhu for their valuable inputs at different stages of the project. This work is partially supported by NSF grants IIS-1527668, CCF-1704883, IIS-1830549, the National Natural Science Foundation of China (Grant Nos. 61761136011, 61532019), the Guangdong Science and Technology Department (Grant No. 2018B010107004), and the Brazilian agency CNPq through the Ciência Sem Fronteiras program.

References

Baier, J. A., and McIlraith, S. 2006. Planning with temporally extended goals using heuristic search. In *ICAPS*, 342–345. AAAI Press.

Bansal, S.; Li, Y.; Tabajara, L. M.; and Vardi, M. Y. 2019. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. *CoRR* abs/1911.08145.

Bohy, A.; Bruyère, V.; Filiot, E.; Jin, N.; and Raskin, J. 2012. Acacia+, a tool for LTL synthesis. In *CAV*.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 100(8):677–691.

Camacho, A.; Baier, J. A.; Muise, C.; and McIlraith, S. A. 2018. Finite LTL synthesis as planning. In *ICAPS*, 29–38. AAAI Press.

Church, A. 1957. Applications of recursive arithmetic to the problem of circuit synthesis. *Institute for Symbolic Logic, Cornell University*.

Cohen, H.; Whaley, J.; Wildt, J.; and Gorogiannis, N. 2014. BuDDy. http://sourceforge.net/p/buddy/.

Dax, C.; Eisinger, J.; and Klaedtke, F. 2007. Mechanizing the powerset construction for restricted classes of ω-automata. In *ATVA*, 223–236. Springer.

De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, 854–860. AAAI Press.

De Giacomo, G., and Vardi, M. 2015. Synthesis for LTL and LDL on finite traces. In *IJCAI*, 1558–1564. AAAI Press.

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. Springer.

Filiot, E.; Jin, N.; and Raskin, J.-F. 2010. Compositional algorithms for LTL synthesis. In *ATVA*, 112–127. Springer.

Filiot, E.; Jin, N.; and Raskin, J.-F. 2011. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design* 39(3):261–296.

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

He, K.; Lahijanian, M.; Kavraki, L. E.; and Vardi, M. Y. 2017. Reactive synthesis for finite tasks under resource constraints. In *IROS*.

He, K.; Wells, A. M.; Kavraki, L. E.; and Vardi, M. Y. 2019. Efficient symbolic reactive synthesis for finite-horizon tasks. In *ICRA*, 8993–8999. IEEE.

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

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

Kupferman, O., and Vardi, M. Y. 1999. Model checking of safety properties. In *CAV*, 172–183. Springer.

Kupferman, O. 2018. Automata theory and model checking. In *Handbook of Model Checking*. Springer. 107–151.

Lahijanian, M.; Almagor, S.; Fried, D.; Kavraki, L. E.; and Vardi, M. Y. 2015. This time the robot settles for a cost: A quantitative approach to temporal logic planning with partial satisfaction. In *AAAI*.

Mazala, R. 2002. Infinite games. In *Automata logics, and infinite games*. Springer. 23–38.

Meyer, P. J.; Sickert, S.; and Luttenberger, M. 2018. Strix: Explicit reactive synthesis strikes back! In *CAV*, 578–586. Springer.

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

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

Tabajara, L. M., and Vardi, M. Y. 2019. Partitioning techniques in LTLf synthesis. In *IJCAI*, 5599–5606. AAAI Press.

Tabakov, D.; Rozier, K. Y.; and Vardi, M. Y. 2012. Optimized temporal monitors for SystemC. *Formal Methods in System Design* 41(3):236–268.

Thomas, W.; Wilke, T.; et al. 2002. *Automata, logics, and infinite games: A guide to current research*, volume 2500. Springer.

Wimmer, R.; Herbstritt, M.; Hermanns, H.; Strampp, K.; and Becker, B. 2006. Sigref–a symbolic bisimulation tool box. In *ATVA*, 477–492. Springer.

Zhu, S.; Tabajara, L. M.; Li, J.; Pu, G.; and Vardi, M. Y. 2017a. A symbolic approach to safety LTL synthesis. In *In Proc. of HVC*.

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