Symbolic LTL_f Synthesis

Abstract

LTL_f synthesis is the process of finding a strategy that satisfies a linear temporal specification over finite traces. An existing solution to this problem relies on a reduction to a DFA game. In this paper, we propose a symbolic framework for LTL_f synthesis based on this technique, by performing the computation over a representation of the DFA as a boolean formula rather than as an explicit graph. This approach enables strategy generation by utilizing the mechanism of boolean synthesis. We implement this symbolic synthesis method in a tool called *Syft*, and demonstrate by experiments on scalable benchmarks that the symbolic approach scales better than the explicit one.

1 Introduction

The problem of synthesis from temporal specifications can be used to model a number of different problems in AI, in particular planning. Linear Temporal Logic, or LTL, is the standard formalism to describe these specifications, but while LTL is classically interpreted over infinite runs, many planning problems have a *finite horizon*, since they assume that execution stops after a specific goal is achieved. This context leads to the emergence of a version of LTL with alternative semantics over finite traces, called LTL_f. Some of the planning problems which can be reduced to LTL_f synthesis include several variants of conditional planning with full observability [Bacchus and Kabanza, 2000; Gerevini *et al.*, 2009]. A general survey of applications of LTL_f in AI and CS can be found in [De Giacomo and Vardi, 2013].

LTL synthesis in infinite-horizon settings has been well investigated in theory since [Pnueli and Rosner, 1989], but the lack of good algorithms for the crucial step of automata determinization is prohibitive for finding practical implementations [Fogarty *et al.*, 2013]. In addition, usual approaches rely on parity games [Thomas, 1995], for which no polynomial-time algorithm is known. In contrast, in the finite-horizon setting the specification can be represented by a finite-state automaton, for which determinization is practically feasible [Tabakov *et al.*, 2012] and the corresponding game can be solved in polynomial time on the number of states and transitions of the deterministic automaton. This opens the possibility for theoretical solutions to be implemented effectively.

A solution to the LTL_f synthesis problem was first proposed in [De Giacomo and Vardi, 2015], based on DFA games. Following this method, an LTL_f specification can be transformed into a DFA with alphabet comprised of propositional interpretations of the variables in the formula. A winning strategy for the game defined by this DFA is then guaranteed to realize the temporal specification.

In this paper, we present the first practical implementation of this theoretical framework for LTL_f synthesis, in the form of a tool called *Syft*. *Syft* follows a symbolic approach based on an encoding of the DFA using boolean formulas, represented as Binary Decision Diagrams (BDDs), rather than an explicit representation through the state graph. We base the choice of a symbolic approach in experiments on DFA construction from an LTL_f specification. We compared between two methods for DFA construction: one symbolic using the tool MONA [Henriksen *et al.*, 1995], receiving as input a translation of the LTL_f specification to first-order logic, and one explicit using SPOT [Duret-Lutz *et al.*, 2016]. Although both methods display limited scalability, the results show that the symbolic construction scales significantly better.

Using a symbolic approach allows us to leverage techniques for *boolean synthesis* [Fried *et al.*, 2016] in order to compute the winning strategy. Our synthesis framework employs a fixpoint computation to construct a formula that expresses the choices of outputs in each state that move the game towards an accepting state. By giving this formula as input to a boolean synthesis procedure we can obtain a winning strategy whenever one exists.

Further experiments comparing the performance of the *Syft* tool with an explicit implementation *E-Syft* again display better scalability for the symbolic approach. However, the results show DFA construction to be the limiting factor in the synthesis process.

2 Preliminaries

2.1 LTL_f Basics

Linear Temporal Logic (LTL) over finite traces, i.e. LTL_f, has the same syntax as LTL over infinite traces introduced in [Pnueli, 1977]. Given a set of propositions P, the syntax of LTL_f formulas is defined as follows:

$$\phi ::= \top \mid \bot \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid X\phi \mid \phi_1 U \phi_2$$

$\top$ and $\bot$ represent *true* and *false* respectively. $a \in P$ is an *atom*, and we define a literal l to be an atom or the negation of an atom. X (Next) and U (Until) are temporal operators. We also introduce their dual operators, namely X_w (Weak Next) and R (Release), defined as $X_w\phi \equiv \neg X \neg \phi$ and $\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$. Additionally, we define the abbreviations $F\phi \equiv \top U \phi$ and $G\phi \equiv \bot R \phi$. Standard boolean abbreviations, such as $\vee$ (or) and $\rightarrow$ (implies) are also used.

A *trace* $\rho = \rho[0], \rho[1], \ldots$ is a sequence of propositional interpretations (sets), in which $\rho[m] \in 2^P$ ($m \geq 0$) is the m-th interpretation of ρ, and $|\rho|$ represents the length of ρ. Intuitively, $\rho[m]$ is interpreted as the set of propositions which are $true$ at instant m. Trace ρ is an *infinite* trace if $|\rho| = \infty$, which is formally denoted as $\rho \in (2^P)^\omega$; otherwise ρ is a *finite* trace, denoted as $\rho \in (2^P)^*$. LTL_f formulas are interpreted over finite traces. Given a finite trace ρ and an LTL_f formula ϕ, we inductively define when ϕ is $true$ for ρ at step i ($0 \leq i < |\rho|$), written $\rho, i \models \phi$, as follows:

- $\rho, i \models \top$;
- $\rho, i \not\models \bot$;
- $\rho, i \models a$ iff $a \in \rho[i]$;
- $\rho, i \models \neg\phi$ iff $\rho, i \not\models \phi$;
- $\rho, i \models \phi_1 \wedge \phi_2$, iff $\rho, i \models \phi_1$ and $\rho, i \models \phi_2$;
- $\rho, i \models X\phi$, iff $i + 1 < |\rho|$ and $\rho, i + 1 \models \phi$;
- $\rho, i \models \phi_1 U \phi_2$, iff there exists j s.t. $i \leq j < |\rho|$ and $\rho, j \models \phi_2$, and for all k, $i \leq k < j$, we have $\rho, k \models \phi_1$.

An LTL_f formula ϕ is $true$ in ρ, denoted by $\rho \models \phi$, if and only if $\rho, 0 \models \phi$.

We next define the LTL_f synthesis problem.

Definition 1 (LTL_f Synthesis)**.** *Let ϕ be an LTL_f formula and $\mathcal{X}, \mathcal{Y}$ be two disjoint atom sets such that $\mathcal{X} \cup \mathcal{Y} = P$. $\mathcal{X}$ is the set of* input variables *and $\mathcal{Y}$ is the set of* output variables*. ϕ is* realizable *with respect to $\langle \mathcal{X}, \mathcal{Y} \rangle$ if there exists a strategy $g : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$, such that for an arbitrary infinite sequence $\lambda = X_0, X_1, \ldots \in (2^{\mathcal{X}})^\omega$ of propositional interpretations over $\mathcal{X}$, we can find $k \geq 0$ such that ϕ is $true$ in the finite trace $\rho = (X_0 \cup g(\epsilon)), (X_1 \cup g(X_0)), \ldots, (X_k \cup g(X_0, X_1, \ldots, X_{k-1}))$.*

Moreover, variables in $\mathcal{X}$ ($\mathcal{Y}$) are called X- (Y-)variables.

2.2 DFA Games

The traditional way of performing LTL_f synthesis is by a reduction to corresponding Deterministic Finite Automaton (DFA) games. According to [De Giacomo and Vardi, 2015], every LTL_f formula can be translated in 2EXPTIME to a DFA which accepts the same language as the formula.

Let the DFA $\mathcal{G}$ be $(2^{\mathcal{X} \cup \mathcal{Y}}, S, s_0, \delta, F)$, where $2^{\mathcal{X} \cup \mathcal{Y}}$ is the alphabet, S is the set of states, s_0 is the initial state, $\delta : S \times 2^{\mathcal{X} \cup \mathcal{Y}} \rightarrow S$ is the transition function, and F is the set of accepting states. A game on $\mathcal{G}$ consists of two players, the controller and the environment. $\mathcal{X}$ is the set of *uncontrollable propositions*, which are under the control of the environment, and $\mathcal{Y}$ is the set of *controllable propositions*, which are under the control of the controller. The DFA game problem is to check the existence of *winning strategy* for the controller and generate it if exists. A strategy for the controller is a function $g : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$, deciding the values of the controllable variables for every possible history of the uncontrollable variables. To define a *winning strategy* for the controller, we use the definition of *winning state* below.

Definition 2 (Winning State)**.** *Given a DFA $\mathcal{G} = (2^{\mathcal{X} \cup \mathcal{Y}}, S, s_0, \delta, F)$, $s \in S$ is a* winning state *if $s \in F$ is an accepting state, or there exists $Y \in 2^{\mathcal{Y}}$ such that, for every $X \in 2^{\mathcal{X}}$, $\delta(s, X \cup Y)$ is a winning state. Such Y is the* winning output *of winning state s.*

Strategy g for the controller is a *winning strategy*, if the initial state s_0 is a *wining state*. In this case, starting from s_0, given an infinite sequence $\lambda = X_0, X_1, X_2, \ldots \in (2^{\mathcal{X}})^\omega$, there is a finite trace $\rho = (X_0 \cup g(\epsilon)), (X_1 \cup g(X_0)), \ldots, (X_k \cup g(X_0, X_1, \ldots, X_{k-1}))$ that ends at an accepting state. After obtaining a DFA from the LTL_f specification, we can utilize the solution to the DFA game for LTL_f synthesis.

Formally, the *winning strategy* can be represented as a deterministic finite transducer, defined as below.

Definition 3 (Deterministic Finite Transducer)**.** *Given a DFA $\mathcal{G} = (2^{\mathcal{X} \cup \mathcal{Y}}, S, s_0, \delta, F)$, a deterministic finite transducer $\mathcal{T} = (2^{\mathcal{X}}, 2^{\mathcal{Y}}, Q, s_0, \varrho, \omega, F)$ is defined as follows:*

- *$Q \subseteq S$ is the set of winning states;*
- *$\varrho : Q \times 2^{\mathcal{X}} \rightarrow Q$ is the transition function such that $\varrho(q, X) = \delta(q, X \cup Y)$ and $Y = \omega(q)$;*
- *$\omega : Q \rightarrow 2^{\mathcal{Y}}$ is the output function such that $\omega(q)$ is a winning output of q.*

2.3 Boolean Synthesis

In this paper, we utilize the *boolean synthesis* technique proposed in [Fried *et al.*, 2016]. The formal definition of the boolean synthesis problem is as follows.

Definition 4 (Boolean Synthesis [Fried *et al.*, 2016])**.** *Given two disjoint atom sets $\mathcal{X}, \mathcal{Y}$ of input and output variables, respectively, and a boolean formula ξ over $\mathcal{X} \cup \mathcal{Y}$, the boolean synthesis problem is to construct a function $\gamma : 2^{\mathcal{X}} \rightarrow 2^{\mathcal{Y}}$ such that, for all $X \in 2^{\mathcal{X}}$, if there exists $Y \in 2^{\mathcal{Y}}$ such that $X \cup Y \models \xi$, then $X \cup \gamma(X) \models \xi$. The function γ is called the* implementation function*.*

We treat boolean synthesis as a black box, using the implementation function construction in the LTL_f synthesis to obtain the output function of the transducer. For more details on boolean synthesis we refer to [Fried *et al.*, 2016].

3 Translation from LTL_f formulas to DFA

Following [De Giacomo and Vardi, 2015], in order to use DFA games to solve the synthesis problem, we need to first convert the LTL_f specification to a DFA. This section focuses on DFA construction. Given an LTL_f formula ϕ, the corresponding DFA can be constructed explicitly or symbolically.

3.1 DFA construction

SPOT [Duret-Lutz *et al.*, 2016] is the state-of-the-art platform for conversion from LTL formulas to explicit Deterministic Büchi Automaton (DBA). The reduction rules from an LTL_f

formula ϕ to an LTL formula ϕ_e are proposed in [De Giacomo and Vardi, 2013], and are already implemented in SPOT. Thus by giving an LTL_f formula to SPOT, it returns the DBA D_e for ϕ_e. D_e can be trimmed to a DFA that recognizes the language of the LTL_f formula ϕ. For more details, we refer to [Dutta *et al.*, 2013; Dutta and Vardi, 2014].

MONA [Henriksen *et al.*, 1995] is a tool that translates from the Weak Second-order Theory of One or Two successors (WS1S/WS2S) to symbolic DFA. First Order Logic (FOL) on finite words, which is a fragment of WS1S, has the same expressive power as LTL_f, so an LTL_f formula ϕ can be translated to a corresponding FOL formula fol_ϕ [De Giacomo and Vardi, 2013]. Taking such a FOL formula fol_ϕ as input, MONA is able to generate the DFA for ϕ.

3.2 Evaluations

It is unnecessary to compare the outputs of SPOT and MONA in terms of size, since both tools return a minimized DFA.The key point is to test them in scalability. LTL and LTL_f have the same syntax, so we construct our benchmarks from 20 basic cases, half of which are realizable, from the LTL literature [Jobstmann and Bloem, 2006]. The number of states of the DFA for each basic case is less than 20.

Since these basic cases are too small to be used individually to evaluate the DFA construction tools, we use a class of *random conjunctions* over basic cases [Daniele *et al.*, 1999]. Note that real specifications typically consist of many temporal properties, whose conjunction ought to be realizable. Formally, a random conjunction formula $RC(L)$ has the form: $RC(L) = \bigwedge_{1 \le i \le L} P_i(v_1, v_2, ..., v_k)$, where L is the number of conjuncts, or the length of the formula, and P_i is a randomly selected basic case (out of the 20 ones). Variables $v_1, v_2, ..., v_k$ are chosen randomly from a set of m candidate variables. Given L and m (the size of the candidate variable set), we generate a formula $RC(L)$ in the following way: (1) Randomly select L basic cases; (2) For each case ϕ, substitute every variable v with a random new variable v' chosen from m atomic propositions. If v is an X-variable in ϕ, then v' is also an X-variable in $RC(L)$. The same applies to the Y-variables.

Each candidate variable may be chosen multiple times, so the number of variables in the formula varies. We generate 50 random formulas for each configuration (L, m), adding up to 4500 instances in total. Formula lengths L range from 1 to 10, and m varies in increments of 50 from 100 to 500. The platform used in the experiments is a computer cluster consisting of 2304 processor cores in 192 Westmere nodes (12 processor cores per node) at 2.83 GHz with 4GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8GB of RAM per core. Time out was set to 120 seconds. Cases that cannot generate the DFA within 120 seconds fail even if the time limit is extended, due to running out of memory.

Here we consider the number of successfully converted cases for scalability evaluation. The results are summarized in Figure 1 and 2, which present the scalability of SPOT and MONA on L and m respectively. As L grows, MONA demonstrates greater scalability, since SPOT cannot handle cases for $L > 4$. We conjecture that the poor scalability on L of SPOT is due to the bad handling of conjunctive goals. In the comparison of scalability on m, MONA is able to solve around twice as many cases than SPOT for each m. Given these results, we adopt MONA for the DFA construction process and pursue a symbolic approach for LTL_f synthesis.

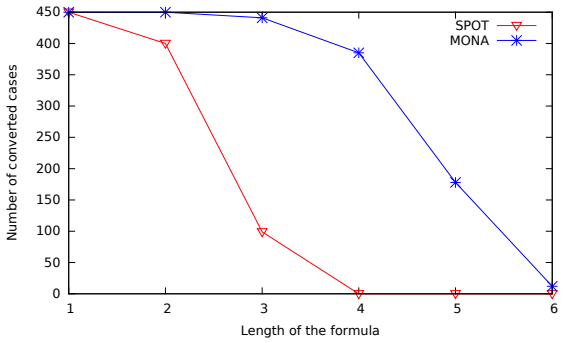

Figure 1: Comparison of scalability between SPOT and MONA on the length of formulas

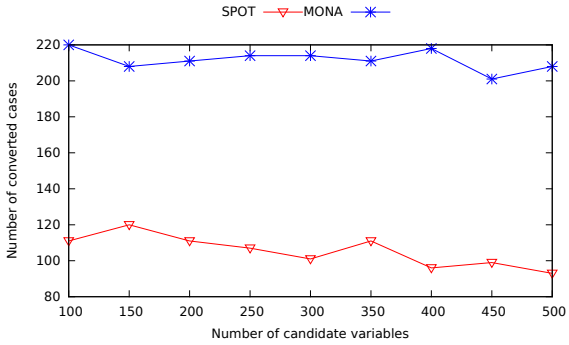

Figure 2: Comparison of scalability between SPOT and MONA on the number of variables

4 Symbolic LTL_f Synthesis

From an explicit automaton, the DFA game can be solved following the approach described in [De Giacomo and Vardi, 2015], by searching the state graph to compute the set of winning states and choosing for each winning state a winning output. The winning states and outputs can then be used to construct a transducer that implements the winning strategy.

To solve a DFA game symbolically, we need a symbolic representation of the automaton $\mathcal{G} = (2^{\mathcal{X} \cup \mathcal{Y}}, S, s_0, \delta, F)$. For that purpose, we define a *symbolic automaton*.

Definition 5 (Symbolic Automaton)**.** *A symbolic automaton $\mathcal{F} = (\mathcal{X}, \mathcal{Y}, \mathcal{Z}, Z_0, \eta, f)$ is defined as follows:*

- *$\mathcal{X}$ and $\mathcal{Y}$ are as defined for $\mathcal{G}$;*
- *$\mathcal{Z}$ is a set of $\lceil \log_2 |S| \rceil$ new propositions such that every state $s \in S$ corresponds to an interpretation $Z \in 2^{\mathcal{Z}}$;*
- *$Z_0 \in 2^{\mathcal{Z}}$ is an interpretation of the propositions in $\mathcal{Z}$ corresponding to the initial state s_0;*
- *$\eta : 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \times 2^{\mathcal{Z}} \to 2^{\mathcal{Z}}$ is a boolean function mapping interpretations X, Y and Z of the propositions of $\mathcal{X}$, $\mathcal{Y}$*

and $\mathcal{Z}$ to a new interpretation Z' of the propositions of $\mathcal{Z}$, such that if Z corresponds to a state $s \in S$ then Z' corresponds to the state $\delta(s, X \cup Y)$;

- *f is a boolean formula over the propositions in $\mathcal{Z}$, such that f is satisfied by an interpretation Z iff Z corresponds to a final state $s \in F$.*

Intuitively, the *symbolic automaton* represents states by propositional interpretations, the transition function by a boolean function and the set of final states by a boolean formula.

Then, to solve the realizability problem we compute a boolean formula w over $\mathcal{Z}$ that is satisfied exactly by those interpretations that correspond to winning states. The specification is realizable if and only if Z_0 satisfies w. To solve the synthesis problem, we compute a boolean function $\tau : 2^{\mathcal{Z}} \rightarrow 2^{\mathcal{Y}}$ such that for any sequence $(X_0, Y_0, Z_0), (X_1, Y_1, Z_1), \ldots$ that satisfies: (1) Z_0 is the initial state; (2) For every $i \geq 0$, $Y_i = \tau(Z_i)$; (3) For every $i \geq 0$, $Z_{i+1} = \eta(X_i, Y_i, Z_i)$ there exists an i such that Z_i satisfies f. In other words, starting from the initial state, for any sequence of uncontrollable variables, if the controllable variables are computed by τ and the next state is computed by η, the play eventually reaches an accepting state.

4.1 Realizability and Synthesis over Symbolic Automaton

We can compute w and τ through a fixpoint computation over two boolean formulas: w_i, over the set of propositions $\mathcal{Z}$, and t_i, over $\mathcal{Z} \cup \mathcal{Y}$. These formulas encode winning states and winning outputs in the following way: every interpretation $Z \in 2^{\mathcal{Z}}$ such that $Z \models w_i$ corresponds to a winning state, and every interpretation $(Z, Y) \in 2^{\mathcal{Z}} \times 2^{\mathcal{Y}}$ such that $(Z, Y) \models t_i$ corresponds to a winning state together with a winning output of that state. When we reach a fixpoint, w_i should encode all winning states and t_i all pairs of winning states and winning outputs.

In the procedure below, we compute the fixpoints of w_i and t_i starting from w_0 and t_0. We assume that we are able to perform basic Boolean operations over the formulas, as well as substitution, quantification and testing for logical equivalence of two formulas.

In the first step of the computation, we initialize $t_0(Z, Y) = f(Z)$ and $w_0(Z) = f(Z)$, since every accepting state is a winning state. Note that t_0 is independent of the propositions from $\mathcal{Y}$, since once the play reaches an accepting state the game is over and we don't care about the outputs anymore. Then we construct t_{i+1} and w_{i+1} as follows: $t_{i+1}(Z, Y) = t_i(Z, Y) \vee (\neg w_i(Z) \wedge \forall X. w_i(\eta(X, Y, Z)))$, $w_{i+1}(Z) = \exists Y. t_{i+1}(Z, Y)$

An interpretation $(Z, Y) \in 2^{\mathcal{Z}} \times 2^{\mathcal{Y}}$ satisfies t_{i+1} if either: (Z, Y) satisfies t_i; or Z was not yet identified as a winning state, and for every input X we can move from Z to an already-identified winning state by setting the output to Y. Note that it is important in the second case that Z has not yet been identified as a winning state, because it guarantees that the next transition will move closer to the accepting states. Otherwise, it would be possible, for example, for t_{i+1} to accept an assignment to Y that moves from Z back to itself, making the play stuck in a self loop.

From t_{i+1}, we can construct w_{i+1} by existentially quantifying the output variables. This means that w_{i+1} is satisfied by all interpretations $Z \in \mathcal{Z}$ that satisfy t_{i+1} for some output, ignoring what the output is. The computation reaches a fixpoint when $w_{i+1} \equiv w_i$ ($\equiv$ denoting logical equivalence). At this point, no more states will be added, and so all winning states have been found. By evaluating w_i on Z_0 we can know if there exists a winning strategy. If that is the case, t_i can be used to compute this strategy. This can be done through the mechanism of boolean synthesis.

By giving t_i as the input formula to a boolean synthesis procedure, and setting $\mathcal{Z}$ as the input variables and $\mathcal{Y}$ as the output variables, we get back a function $\tau : 2^{\mathcal{Z}} \rightarrow 2^{\mathcal{Y}}$ such that $(Z, \tau(Z)) \models t_i$ if and only if there exists $Y \in 2^{\mathcal{Y}}$ such that $(Z, Y) \models t_i$.

Using τ, we can define a *symbolic transducer* $\mathcal{H}$ corresponding to the winning strategy of the DFA game.

Definition 6 (Symbolic Transducer)**.** *Given a symbolic automaton $\mathcal{F} = (\mathcal{X}, \mathcal{Y}, \mathcal{Z}, Z_0, \eta, f)$ and a function $\tau : 2^{\mathcal{Z}} \rightarrow 2^{\mathcal{Y}}$, the symbolic transducer $\mathcal{H} = (\mathcal{X}, \mathcal{Y}, \mathcal{Z}, Z_0, \zeta, \tau, f)$ is as follows:*

- *$\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, Z_0 and f are as defined for $\mathcal{F}$;*
- *τ is as defined above;*
- *$\zeta : 2^{\mathcal{Z}} \times 2^{\mathcal{X}} \rightarrow 2^{\mathcal{Z}}$ is the* transition function *such that $\zeta(Z, X) = \eta(X, Y, Z)$ and $Y = \tau(Z)$.*

Note that τ selects a single winning output for each winning state. Such a transducer is a solution to the DFA game, and therefore to the LTL_f synthesis problem.

5 Implementation

We implemented the symbolic synthesis procedure described in Section 4 in a tool called *Syft*, using Binary Decision Diagrams (BDDs) to represent the boolean formulas. For comparison, we also implemented the explicit version in a tool called *E-Syft*. Both were implemented in C++11, and *Syft* uses CUDD-3.0.0 as the BDD library.

Each tool consists of two parts: DFA construction and the synthesis procedure. Based on the evaluations of the performance of DFA construction in Section 3, we adopt MONA to construct the DFA to be given as input to the synthesis procedure. The DFA is given by MONA as a *Shared Multi-terminal BDD* (ShMTBDD) [Bryant, 1992; Biehl *et al.*, 1996], which represents the function $\delta : S \times 2^{\mathcal{X} \cup \mathcal{Y}} \rightarrow S$. A ShMTBDD is a binary decision diagram with $|S|$ roots and m terminal nodes ($m \leq |S|$) representing states in the automaton. Formally speaking, $\delta(s, X \cup Y) = s'$ is a transition in the DFA if and only if starting from the root representing state s and evaluating the interpretation $X \cup Y$ leads to the terminal representing state s'. To evaluate an interpretation $X \cup Y$ on a ShMTBDD, take the high branch in every node labeled by a variable $v \in X \cup Y$ and the low branch otherwise. We next describe how we preprocess the DFA given by MONA.

5.1 Preprocessing the DFA of MONA

From ShMTBDD to Explicit DFA. Each root in an ShMTBDD corresponds to an explicit state in the DFA.

Moreover, a root of the ShMTBDD includes the information about if the state is initial or accepting, thus enabling s_0 and F to be easily extracted for the explicit DFA. To construct the transition function, we enumerate all paths in the ShMTBDD, each path from s (root) to t (terminal node) corresponding to one transition from state s to t in the DFA. Note that the size of the explicit DFA may be exponential on the size of the ShMTBDD.

From ShMTBDD to BDD. Following Section 4, we can construct a symbolic automaton in which the transition function η is in form of a (multi-rooted) BDD that describes a boolean function $\eta : 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \times 2^{\mathcal{Z}} \to 2^{\mathcal{Z}}$. Thus we need to first generate the BDD for the given ShMTBDD.

The basic idea is as follows: (1) From the ShMTBDD of δ, construct a Multi-Terminal BDD (MTBDD) for δ' : $2^{\mathcal{Z}} \times 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \to S$ with $\lceil \log_z |S| \rceil$ new boolean variables encoding the states, where every path through the state variables, representing an interpretation Z_s, leads to the node under the root s in the ShMTBDD. Then, for each transition in δ, there exists an equivalent transition in δ'. (2) Decompose the MTBDD into a sequence of BDDs $\mathcal{B} = \langle B_0, B_1, ..., B_{n-1} \rangle$, $n = \lceil \log_z |S| \rceil$, where each B_i, when evaluated on an interpretation $(X \cup Y \cup Z)$, computes the i-th bit in the binary encoding of state $\delta'(X, Y, Z)$.

The idea of splitting the ShMTBDD into BDDs is illustrated on Figure 3. As shown in this example, bits b_0, b_1 are used to denote the four states s_0, s_1, s_2, s_3. In step (1), root s_0 is substituted by Z_{s_0} that corresponds to the formula $(\neg b_0 \wedge \neg b_1)$. After replacing all roots with corresponding interpretations, the MTBDD is produced. In step (2), s_0, s_1, s_2, s_3 can be represented by $00, 01, 10, 11$ respectively, where b_0 denotes the leftmost bit. Bit b_0 for both s_0 and s_1 is 0. So by forcing all paths that proceed to terminals s_0 and s_1 in the MTBDD to reach terminal node 0, and all paths to terminals s_2 and s_3 to reach terminal node 1, BDD B_0 is generated. BDD B_1 is constructed in an analogous way for bit b_1.

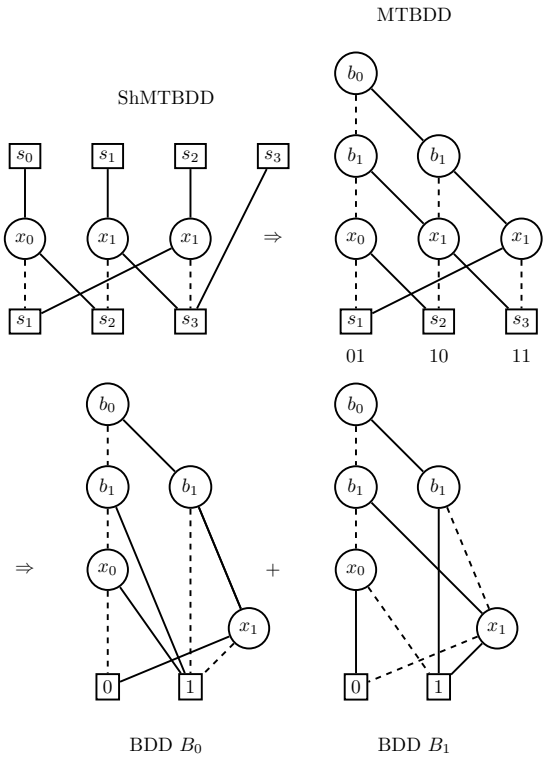

Figure 3: Transformation from ShMTBDD to BDD

5.2 Implementing the Synthesis Algorithms

Explicit Synthesis. Following [De Giacomo and Vardi, 2015], the main algorithm for explicit synthesis is as follows: starting from accepting states in F, iteratively expand the set of winning states. For every state s that is not yet a winning state, if we find an assignment Y s.t. for all assignments X of input variables, $\delta(s, X \cup Y)$ is a winning state, then Y is set to be the winning output of the new winning state s. All assignments of Y have to be enumerated in the worst case. Fixpoint checking is accomplished by checking that no new winning state was added during each iteration. The transducer is generated according to Definition 3.

Symbolic Synthesis. The input here is a symbolic automaton $\mathcal{F}$, in which the transition relation η is represented by a sequence $\mathcal{B} = \langle B_0, B_1, ..., B_{n-1} \rangle$ of BDDs, where each B_i corresponding to bit b_i, and the formula f for the accepting states, is represented by BDD B_f. We separate the DFA game into two phases, *realizability* and *strategy construction*.

Following the theoretical framework in Section 4, from the accepting states B_f, we construct two BDD sequences $\mathbf{T}$ and $\mathbf{W}$, such that $\mathbf{T} = \langle B_{t_0}, B_{t_1}, ..., B_{t_{i-1}}, B_{t_i} \rangle$ and $\mathbf{W} = \langle B_{w_0}, B_{w_1}, ..., B_{w_{i-1}}, B_{w_i} \rangle$, where B_{t_i} and B_{w_i} are the BDDs of the boolean formulas t_i and w_i respectively. The fixpoint computation terminates as soon as $B_{w_{i+1}} \equiv B_{w_i}$.

Our implementation uses the CUDD BDD library [Somenzi, 2016], where fundamental BDD operations are provided. In realizability checking, $B_{t_{i+1}}$ is constructed from B_{w_i} by first substituting each bit b_i of the binary encoding of the state with the corresponding BDD B_i. The BDD operation *Compose* is used for such substitution. Universal quantifier elimination of each variable $x \in \mathcal{X}$ is performed via the *UnivAbstract* BDD operation. From $B_{t_{i+1}}$, applying existential quantifier elimination to each variable $y \in \mathcal{Y}$ via the BDD operation *ExistAbstract* yields the BDD $B_{w_{i+1}}$. For fixpoint checking, we use canonicity of BDDs, which reduces equivalence checking to constant-time BDD equality comparison. To check realizability, the fixpoint BDD B_{w_i} is evaluated on interpretation Z_0 of state variables, returning 1 if realizable.

Since version 3.0.0, CUDD includes a built-in boolean synthesis method *SolveEqn*, which we can use to generate the function τ. From τ, the symbolic transducer can be constructed according to Definition 6.

6 Experiments

To evaluate the performance of the symbolic and explicit tools, we carried out experiments on the same platform as Section 3. For the synthesis experiments, besides the original 20 basic cases we also collected 80 instances from the

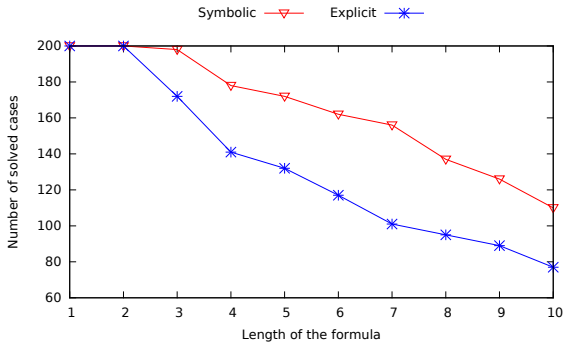

Figure 4: Comparison of scalability of Symbolic against Explicit on the length of formula

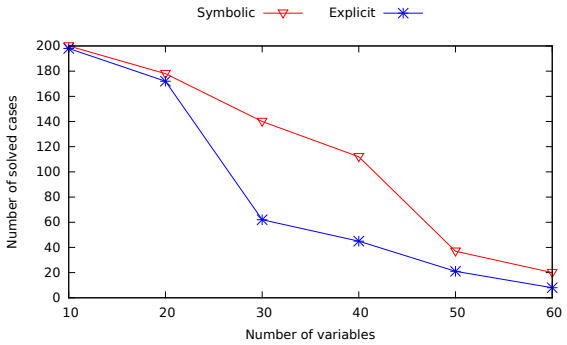

Figure 5: Comparison of scalability of Symbolic against Explicit on the number of variables

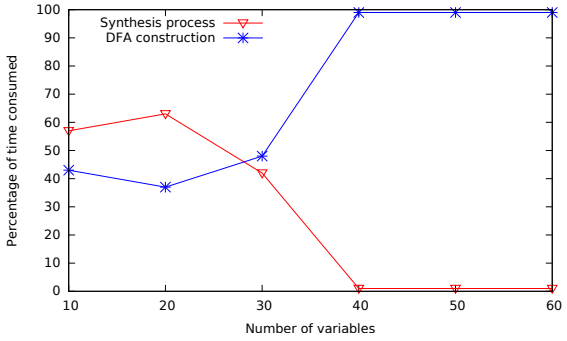

Figure 6: Comparison of scalability of Symbolic against Explicit on the number of variables

LTL synthesis tool *Acacia+* [Bohy *et al.*, 2012], making 100 cases in total. In this section, L denotes the length of the formulas and N denotes the approximate number of variables. Due to the construction rules of the formula, as described in Section 3, each variable is chosen randomly and may be chosen multiple times. Thus, the exact number of variables in a formula for a given N varies in the range $(N-5, N+5]$.

Scalability on the length of formulas. We evaluated the scalability of *Syft* and *E-Syft* on 2000 benchmarks where the formula length L ranges from 1 to 10 and $N = 20$, as MONA is more likely to succeed in constructing the DFA for this value of N. Each L accounts for 200 of the 2000 benchmarks. Figure 4 shows the number of cases the tools were able to solve for each L. As can be seen, the symbolic method can handle a larger number of cases than the explicit one, which demonstrates that the symbolic method outperforms the explicit approach in scalability on the length of the formula.

Scalability on the number of variables. In this experiment, we compared the scalability over the number of variables N, which varies from 10 to 60, where L is fixed as 5. As can be seen in Figure 5, for smaller cases the two approaches behave similarly, since they succeed in almost all cases. The same happens for larger cases, because MONA cannot construct the DFAs. For intermediate values the difference is more noticeable, showing better performance by the symbolic tool. The number of cases that the explicit method can solve sharply declines when $N = 30$. However, the symbolic tool can handle more than 40 variables. Both methods tend to fail for $N > 50$ at the DFA conversion stage.

Synthesis vs DFA Construction. In this experiment we studied the effect of N on the time consumed by DFA construction and synthesis. The percentage of time consumed by each is shown in Figure 6. We observe that for large cases, DFA construction dominates the running time. As the size of the DFA increases, DFA construction takes significantly longer, while synthesis time increases more slowly, widening the gap between the two. This result allows us to conclude that the critical performance limitation of synthesis is the DFA construction process rather than synthesis itself.

Discussion The symbolic synthesis method scales better than the explicit approach both on the length of LTL_f formulas and the number of variables. However, the performance of the symbolic method highly relies on the DFA construction process, making this the bottleneck of LTL_f synthesis.

7 Conclusion

We presented here the first realization of a symbolic approach for LTL_f synthesis, based on the theoretical framework of DFA games. Our experiments on DFA construction and synthesis have shown that a symbolic approach to this problem has advantages over an explicit one. In both cases, however, the limiting factor for scalability was DFA construction. When the DFA could be constructed, the symbolic procedure was able to synthesize almost all cases, but for larger numbers of variables DFA construction is not able to scale.

These observations suggest the need for more scalable methods of symbolic DFA construction. A promising direction would also be to develop techniques for performing synthesis "on the fly" during the construction of the DFA, rather than waiting for the entire automaton to be constructed before initiating the synthesis procedure.

In [De Giacomo and Vardi, 2013] an alternative formalism for finite-horizon temporal specifications is presented in the form of Linear Dynamic Logic over finite traces, or LDL_f. LDL_f is strictly more expressive than LTL_f, but is also expressible as a DFA. Therefore, given a procedure to perform the conversion from LDL_f to DFA, our approach can be used in the same way. This would allow synthesis to be performed over a larger class of specifications.

References

[Bacchus and Kabanza, 2000] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2), 2000.

[Biehl *et al.*, 1996] Morten Biehl, Nils Klarlund, and Theis Rauhe. Mona: decidable arithmetic in practice (demo). In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, Uppsala, LNCS 1135*, 1996.

[Bohy *et al.*, 2012] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012.

[Bryant, 1992] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[Daniele *et al.*, 1999] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 1999.

[De Giacomo and Vardi, 2013] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proc. of IJCAI*, 2013.

[De Giacomo and Vardi, 2015] Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on finite traces. In *Proc. of IJCAI*, 2015.

[Duret-Lutz *et al.*, 2016] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA'16*. Springer, 2016.

[Dutta and Vardi, 2014] Sonali Dutta and Moshe Y. Vardi. Assertion-based flow monitoring of systemc models. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014*, pages 145–154. IEEE, 2014.

[Dutta *et al.*, 2013] Sonali Dutta, Moshe Y. Vardi, and Deian Tabakov. CHIMP: A tool for assertion-based dynamic verification of systemc models. In *Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013.*, 2013.

[Fogarty *et al.*, 2013] Seth Fogarty, Orna Kupferman, Moshe Y. Vardi, and Thomas Wilke. Profile trees for büchi word automata, with application to determinization. In *Proceedings Fourth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2013, Borca di Cadore, Dolomites, Italy, 29-31th August 2013.*, pages 107–121, 2013.

[Fried *et al.*, 2016] Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi. Bdd-based boolean functional synthesis. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Springer, 2016.

[Gerevini *et al.*, 2009] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6), 2009.

[Henriksen *et al.*, 1995] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[Jobstmann and Bloem, 2006] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*. IEEE Computer Society, 2006.

[Pnueli and Rosner, 1989] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. pages 46–57, 1977.

[Somenzi, 2016] Fabio Somenzi. Cudd: Cu decision diagram package 3.0.0. universiy of colorado at boulder. 2016.

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

[Thomas, 1995] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.