# Efficient Translation of Safety LTL to DFA using Symbolic Automata Learning and Inductive Inference *

Georgios Giantamidis[1,2] ✉, Stylianos Basagiannis[1], and Stavros Tripakis[3]

[1] United Technologies Research Centre Ireland, Cork, Ireland,
GiantaGE@utrc.utc.com
[2] Aalto University, Otaniemi, Finland
[3] Northeastern University, Boston, MA, USA

**Abstract.** Safety LTL properties are ubiquitous in the verification of safety critical systems. There is already evidence that translating safety properties into DFA rather than Büchi automata results in faster verification times. Conventional translation strategies can in some cases use unnecessarily large amounts of resources. We develop a symbolic adaptation of the $L^*$ active learning algorithm tailored to efficiently translate safety LTL properties into symbolic DFA. We demonstrate how an inductive inference procedure can be used to provide additional input to the algorithm that greatly improves performance for certain important families of properties. For completeness, we also provide an outline and examples of how such a procedure can be implemented. Finally, we compare with state of the art LTL translators and provide experimental evidence where our approach significantly outperforms conventional translation strategies.

**Keywords:** Linear Temporal Logic · Safety Properties · Automata Learning · Symbolic Automata · Inductive Inference

## 1 Introduction & Motivation

Safety properties are pervasive in model based design. Informally, they capture the notion that 'nothing bad should ever happen', which, in turn, can be used to express a great variety of requirements of safety critical systems. A widespread formalism that can be used to describe safety properties is *Safety LTL* (Linear Temporal Logic) [28]. Safety LTL specifications can be used in a variety of ways in the model based design process: They can be used for formal verification of the system, for runtime monitoring during testing, for generating test-cases and, even before any system model is created, satisfiability tests can be performed on them to reveal potential inconsistencies in the original requirements.

* This work was partially supported by the Irish Development Agency (IDA) for UTRC Ireland related to Network of Excellence in Aerospace Cyber Physical Systems.

An important step in the procedures mentioned above is translation of LTL formulas into automata. In particular, it is possible to translate safety LTL formulas to Deterministic Finite Automata (DFA), which is generally desirable, as it has been shown to reduce verification times [26]. In an agile, continuous integration workflow where during prototyping every small change in the requirements triggers a cascade of testing and verification actions that must be performed as fast as possible to keep iteration times low, efficient translation of safety LTL properties into DFA is of paramount importance.

The problems of translating LTL to automata and specifically safety LTL to DFA have received a lot of attention over the years, and while the worst case theoretical complexity w.r.t number of states is exponential on the formula length for non-deterministic automata and doubly exponential on the formula length for deterministic automata, approaches that perform quite well in practice have been developed [21, 3, 19, 15, 8]. Such approaches are generally based on syntactic manipulation of the LTL formula and typically construct an automaton the states of which correspond to subformulas of the original formula, which can subsequently be determinized/minimized. One major drawback of these approaches is that this intermediate automaton can be considerably larger than the final result, which can lead to unnecessarily excessive resource consumption during translation. Another limitation is that, to the best of our knowledge, existing implementations of such approaches cannot take into account a priori knowledge about the target automaton that might be available.

In this work, we present a novel approach for safety LTL to symbolic DFA (SDFA) translation that overcomes both these limitations. Specifically:

- We develop a novel algorithm for syntactically safe LTL to SDFA translation.
- Our algorithm returns a minimal (w.r.t. number of states) SDFA and all intermediately constructed SDFA contain strictly fewer states than the result.
- Our algorithm can be extended to take into account a priori information about the target automaton, which results in significant performance boost.
- We provide an outline and examples of how an inductive inference procedure that provides said a priori information can be implemented.
- We provide a prototype implementation and experimental evidence that the proposed approach (i) behaves comparably with state of the art tools on randomly generated formulas and literature benchmarks and (ii) significantly outperforms the state of the art in certain important property families (even without a priori information about the target automaton).

In section 2 we summarize related work on (safety) LTL to DFA translation and (symbolic) automata learning. In section 3 we introduce the necessary preliminary concepts and algorithms. In section 4 we describe the proposed algorithm and its properties. In section 5 we present our experimental evaluation results. Finally, in section 6 we conclude with some ideas for future work.

## 2 Related Work

The more general problem of translating LTL to Omega (e.g. Büchi, Rabin etc.) automata has been studied extensively before [3, 19, 15, 8]. The state of the art here is Spot [3] and, more recently, Rabinizer [19]. The problem of translating safety LTL to DFA has also received a great amount of attention [21, 16, 20]. This is justified by the fact that using deterministic automata can improve verification times [26]. To the best of our knowledge [21] is the state of the art on translators specialized to turn safety LTL to DFA, hence we compare against it in our experimental evaluation. Spot and Rabinizer are also able to generate deterministic automata if requested, therefore we compare against them as well.

Automata learning and in general grammatical inference is a field that has received a lot of attention over the years [12]. Algorithms here generally fall into two categories, passive learning (learning from examples) and active learning (learning with queries). Symbolic automata learning is an area that recently received some attention [13, 22]. Variations of this problem have been studied earlier as well [17]. Note that, while we do not claim to improve the state of the art in symbolic automata learning, our extension of $L^*$ makes specific assumptions about the system to be learned, which enable a more efficient approach than using a generic learning algorithm.

## 3 Preliminaries

### 3.1 Linear Temporal Logic

*Linear Temporal Logic* (LTL) [24] is a widespread formalism used in model checking based formal verification. The typical automata theoretic model checking approach requires the negation of the LTL expressed property to first be transformed into an automaton on infinite words, for example a Büchi automaton. Then, this automaton is composed with an automaton representing the system, and the resulting product is checked for emptiness.

LTL properties can be classified into two broad categories: *safety* properties and *liveness* properties [4]. Informally, safety properties state that 'something bad never happens', while liveness properties state that 'something good eventually happens'.

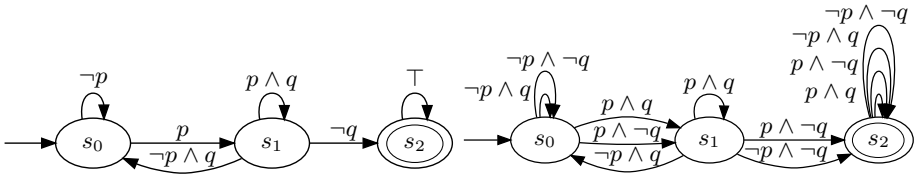
**Syntactic Safety Subset of LTL** Safety LTL properties can also be characterized syntactically. Any property built out of operators in the *syntactic safety subset* of LTL is guaranteed to be a safety property. Specifically, every propositional formula (i.e. a formula built of atomic propositions, $\neg$, $\wedge$ and $\vee$) is a syntactically safe formula, and if formulas $f$ and $g$ are syntactically safe, so are formulas $f \wedge g$, $f \vee g$, $Gf$, $Xf$, $fWg$, where $G$, $X$, and $W$ are, respectively, the *globally*, *next* and *weak until* LTL operators. We omit the formal definition of the semantics of these operators and refer the reader to [28] which we follow in this work. Note that it is possible to express any safety LTL property by only using the syntactic safety subset of LTL.

**Bad Prefixes** Every safety LTL formula $\phi$ can be translated into a DFA which accepts all *bad prefixes* of $\phi$. A bad prefix of $\phi$ is a finite trace $\sigma$ such that all infinite continuations of $\sigma$ violate $\phi$. Kupferman and Vardi [20] further classify safety LTL formulas as *intentionally safe*, *accidentally safe* and *pathologically safe* based on the *informativeness* of their bad prefixes. Intuitively, accidentally safe and pathologically safe formulas contain some redundancy; for example, $G(p \vee X(q \wedge \neg q))$ is accidentally safe, $G(p \vee F(q \wedge \neg q))$ is pathologically safe, and both are equivalent to $Gp$, which is intentionally safe. Since it is possible to write any safety LTL formula as an intentionally safe formula, we will only consider intentionally safe formulas here (however, note that our algorithm handles accidentally safe formulas as well, but not pathologically safe ones, since these do not belong in the syntactic safety subset of LTL).

### 3.2 Symbolic DFA and Symbolic Traces

In this work, we use the definition of symbolic automata from [11]. Symbolic DFA are able to encode the state machine described by a DFA in a more succinct way, by means of allowing predicates drawn from a boolean algebra to compactly represent transitions between states.

Figure 1 shows two monitors for the safety property $G(p \rightarrow Xq)$ expressed as both an SDFA and a DFA ($\top$ as a transition label stands for 'true'). Notice that while the number of states is the same in the two versions, the number of transitions can generally differ drastically. This is very important for the performance of the learning algorithm we employ, as reducing the number of transitions also reduces the amount of book-keeping needed.




(a) SDFA (b) DFA

Fig. 1: Monitors for $G(p \rightarrow Xq)$

Given a set of atomic propositions $\{p_1, p_2, \ldots, p_m\}$ we define a *finite symbolic trace* to be a finite sequence $a_1 \cdot a_2, \cdots, a_n$, where each $a_i$ is either $\top$ or a conjunction of literals (a literal being an atomic proposition, $p_i$, potentially negated).

### 3.3 Active Automata Learning

Our approach is based on Angluin's $L^*$ algorithm for active automata learning [6]. In this setting, a *learner* tries to identify an automaton by submitting queries to a *teacher*. These can be *membership queries*, where the learner submits a

word and gets back an 'accept' or 'reject' answer, or *equivalence queries*, where a hypothesis automaton is submitted and either the process ends with success or a counterexample is generated which drives more queries.

There are four important data structures involved in a run of the $L^*$ algorithm: A $RED$ set of words that represent state candidates for the learned automaton, a $BLUE$ set of words that correspond to 1-step successors of states in $RED$, a set of suffixes, $SFX$ that are used to distinguish states in $RED \cup BLUE$ and the *observation table*, $OBS$, which stores information about words in $RED$ and $BLUE$ w.r.t. their behavior on the suffixes in $SFX$. The set $RED \cup BLUE$ is prefix closed and the set $SFX$ is suffix closed. The rows of $OBS$ are labeled by states in $RED \cup BLUE$ and its columns by elements of $SFX$. The entry of $OBS$ at row $w \in RED \cup BLUE$ and column $s \in SFX$ represents the result of the membership query for the word $w \cdot s$, where dot denotes concatenation. In other words, if the target automaton accepts the word $w \cdot s$, then $OBS(w, s)$ is set to 1, otherwise it is set to 0. We say that $p, q \in RED \cup BLUE$ are $SFX$-equivalent if $\forall s \in SFX : OBS(p, s) = OBS(q, s)$. Otherwise, we say that $p$ and $q$ are $SFX$-distinct. We say that $OBS$ is (i) *complete* if $\forall w \in RED \cup BLUE,\ s \in SFX : OBS(w, s)$ is set to either 1 or 0, (ii) *closed* if $\forall b \in BLUE\ \exists r \in RED : r, b$ are $SFX$-equivalent, and (iii) *consistent* when $\forall p, q \in RED :$ if $p, q$ are $SFX$-equivalent then their 1-step successors $p \cdot \alpha, q \cdot \alpha$ are also $SFX$-equivalent, for each letter $\alpha$ in the alphabet.

A brief description of the algorithm follows:

1. Initially, $RED$ and $SFX$ only contain the empty word, and $BLUE$ contains the 1-letter successors of the empty word.
2. Membership queries are used to make $OBS$ complete, closed and consistent, promoting states from $BLUE$ to $RED$ as needed (to enforce closedness), updating $BLUE$ to include the 1-step successors of any new $RED$ states, as well as potentially adding elements in $SFX$ (to enforce consistency).
3. Once $OBS$ is complete, closed and consistent, a hypothesis DFA is constructed.
4. An equivalence query is performed to check whether the hypothesis is correct.
5. (a) If the hypothesis is incorrect, we obtain a counterexample in the form of a word on which the hypothesis and the target behave differently. We add the counterexample and all its prefixes in $RED$ and go to step 2.
   (b) Otherwise, we have found the target automaton and we are done.

The $L^*$ algorithm is guaranteed to terminate and yield a minimal automaton after at most $n$ equivalence queries and a number of membership queries bounded by a polynomial quadratic on $n$ and linear on $m$, where $n$ is the number of states of the learned machine and $m$ the maximum length of any counterexample word returned by the teacher.

Several variants of the algorithm have been proposed over the years that improve on the original algorithm and extend it to other formalisms. In our implementation we use a variant where the counterexample and all of its suffixes are added in $SFX$ instead of having $RED$ updated as mentioned in step 5a above, as done in [27]. We chose to do so because this introduces a new important

invariant: throughout the algorithm, all states in *RED* are pairwise *SFX*-distinct. In turn, this makes constructing the hypothesis DFA easier: we can simply collect all states in *RED* without worrying that one of them may be equivalent to another. Another nice property of this variant is that *OBS* is guaranteed to be consistent throughout the algorithm, which allows us to skip consistency checking and enforcement in step 2 above.

## 4 Proposed Approach

**Problem Definition & Approach Overview** The problem we are solving can be formulated as follows: 'Given a syntactically safe LTL formula $\Phi$, construct a minimal (w.r.t number of states) symbolic DFA that accepts all bad prefixes of $\Phi$ and nothing else'. Note that the returned DFA must accept the *bad* prefixes of $\Phi$, i.e. all finite traces satisfying $\neg\Phi$. The proposed algorithm is a symbolic extension of Angluin's $L^*$ tailored for LTL to SDFA translation. Membership queries are performed by recursive traversal of the LTL formula itself, as explained in Section 4.2 that follows. As for equivalence queries, we employ a symbolic model checker as explained in Section 4.3. Our method is *symbolic* in the sense that we use a symbolic alphabet (set of predicates) for the learned DFA and employ a lazy alphabet refinement strategy in which we begin with a single, all-encompassing predicate, $\top$, and gradually refine it as needed.

### 4.1 Safety LTL on Finite Symbolic Traces

In order to be able to perform membership queries, it is imperative that we define a semantics for syntactically safe LTL on finite symbolic traces that will allow us to identify bad prefixes. We introduce a four-value semantics where evaluating a formula on a symbolic trace can yield a value of *True*, *False*, *Unknown* or *Refine(proposition, index)*, the last one being parametrized by the proposition that needs refinement and the position in the symbolic trace this needs to happen.

Let $\sigma := a_1 \cdot a_2 \cdots a_n$ be a finite symbolic trace of length $n$ and $\Phi$ a syntactically safe LTL formula. The function `eval`$(\Phi, \sigma, i)$, which returns the evaluation of $\Phi$ on the suffix of $\sigma$ that begins with $a_i$ is defined as follows:

```
eval(p, σ, i) :=
  if ¬p is a conjunct in a_i: return False
  if  p is a conjunct in a_i: return True
  return Refine(p, i)

eval(⊤, σ, i) := return True
eval(⊥, σ, i) := return False

eval(φ ∨ ψ, σ, i) :=
  let e_φ = eval(φ, σ, i), e_ψ = eval(ψ, σ, i)
  if either e_φ or e_ψ is True: return True
  if e_φ (resp. e_ψ) is False: return e_ψ (resp. e_φ)
  if e_φ (resp. e_ψ) is Unknown: return e_ψ (resp. e_φ)
  # now, both e_φ and e_ψ are refinement requests
  if e_φ.index > e_ψ.index: return e_φ
  return e_ψ

eval(φ ∧ ψ, σ, i) :=
  let e_φ = eval(φ, σ, i), e_ψ = eval(ψ, σ, i)
```

```
  if either e_φ or e_ψ is False: return False
  if e_φ (resp. e_ψ) is True: return e_ψ (resp. e_φ)
  if e_φ (resp. e_ψ) is Unknown: return e_ψ (resp. e_φ)
  # now, both e_φ and e_ψ are refinement requests
  if e_φ.index > e_ψ.index: return e_φ
  return e_ψ

eval(¬φ, σ, i) :=
  let e_φ = eval(φ, σ, i)
  if e_φ is True: return False
  if e_φ is False: return True
  return e_φ

eval(φWψ, σ, i) :=
  return eval(ψ ∨ (φ ∧ X(φWψ)), σ, i)

eval(Xφ, σ, i) :=
  if i ≥ n: return Unknown
  return eval(φ, σ, i + 1)

eval(Gφ, σ, i) :=
  return eval(φ ∧ XGφ, σ, i)
```

where $p$ is an atomic proposition and $\phi$, $\psi$ are syntactically safe LTL formulas.

### 4.2 Membership Queries & Lazy Alphabet Refinement

Given the semantics defined in 4.1, we are now able to explain how membership queries work. Recall that when the $L^*$ algorithm submits a membership query it receives an 'accept' or 'reject' answer. In our case, we want a membership query $\texttt{mem\_q}(\Phi, \sigma)$ to return 'accept' iff the symbolic trace $\sigma$ is a bad prefix for the formula $\Phi$. Therefore, if the result of $\texttt{eval}(\Phi, \sigma, 1)$ is $False$, $\texttt{mem\_q}(\Phi, \sigma)$ returns 1 (accept). If the result of $\texttt{eval}(\Phi, \sigma, 1)$ is $True$ or $Unknown$, $\texttt{mem\_q}(\Phi, \sigma)$ returns 0 (reject). In the case $\texttt{eval}(\Phi, \sigma, 1)$ returns a refinement request, the symbolic trace will be duplicated with one copy now containing the positive literal and the other copy the negative literal of the proposition for which refinement was requested, and two separate membership queries will be issued subsequently.

During a run of the $L^*$ algorithm, whenever a word is added to *RED*, its successors with all letters of the alphabet are added to *BLUE* (this also happens during initialization). This means that if we have a formula containing 10 atomic propositions, every time a word is added to *RED*, $2^{10} = 1024$ words will be added to *BLUE*. However, there is a high chance (depending on the formula, of course) that many of these entries actually represent the same state, which implies that a lot of time can potentially be wasted on membership queries that provide essentially the same information. What we do to address this issue is add, instead, a single entry to *BLUE* that symbolically represents all 1-step successors of the state added to *RED*. For example, if the word added to red is $p \cdot p \wedge q$, we add to *BLUE* the word $p \cdot p \wedge q \cdot \top$. Whether the latter actually corresponds to different states will be revealed later, as membership queries are submitted. Suppose, for example, that the algorithm issues the query $\texttt{mem\_q}(G(p \rightarrow Xq), p \cdot p \wedge q \cdot \top)$ or, equivalently, $\texttt{mem\_q}(G(\neg p \vee Xq), p \cdot p \wedge q \cdot \top)$. Since $p$ holds at step 2, but we do not know what happens to $q$ at step 3, a refinement request for $q$ at position 3 is issued. Then, the word will be split accordingly and the following two membership queries will be performed: $\texttt{mem\_q}(G(\neg p \vee Xq), p \cdot p \wedge q \cdot q)$, $\texttt{mem\_q}(G(\neg p \vee Xq), p \cdot p \wedge q \cdot \neg q)$, the former of which returns $Unknown$ and the latter $False$. And since we now know that $p \cdot p \wedge q \cdot \neg q$ is a bad prefix for the formula, the corresponding cell in *OBS* will be set to 1. Accordingly, the cell

corresponding to $p \cdot p \wedge q \cdot q$ will be set to 0, which would also be the case for a result of value $True$.

The astute reader may wonder here why we need both values $True$ and $Unknown$ if the purpose of a membership query is to detect whether a word is a bad prefix or not. The answer has to do with the behavior of $True$ and $Unknown$ w.r.t. refinement requests: $True$ and $False$ have 'priority' over $Refine$, which in turn has 'priority' over $Unknown$, as can be seen in the definitions of `eval` for $\phi \vee \psi$ and $\phi \wedge \psi$ above. This ensures that on one hand we perform refinement when necessary, but on the other hand do not refine without a real need to do so. For the same reason, we heuristically perform first the refinement request that refers to the latest position in the trace.

### 4.3 Equivalence Queries

Equivalence queries are implemented by employing NuSMV [9], a symbolic model checker. Whenever an equivalence query needs to be issued, a hypothesis automaton is constructed, encoded in the language of NuSMV and then model checked against the following properties:

$$\texttt{LTLSPEC}\ \ \Phi \leftrightarrow G\ \ state \neq accept \qquad\qquad \texttt{LTLSPEC}\ \ \neg\Phi \leftrightarrow F\ \ G\ \ state = accept$$

where $\Phi$ is the LTL formula we want to translate, *state* is a variable holding the current state of the encoded hypothesis automaton and *accept* is a value denoting its (unique) accepting state. Note that in order to perform the above checks, NuSMV does *not* internally translate the LTL formula into an automaton (which would defeat the purpose of the proposed algorithm). Rather, it translates the LTL formula into a CTL one with added fairness constraints and then applies a symbolic model checking algorithm [25, 10, 14].

Counterexample handling is a bit involved, since what NuSMV returns is a description of an infinite trace. What we do is we lazily enumerate all finite prefixes of this infinite counterexample in order of increasing length and pick the first one that reveals a problem in the hypothesis. We opted for choosing the shortest possible counterexample because small counterexample length means less subsequent membership queries, and while longer counterexamples might reveal more new states, this is not guaranteed.

Another interesting direction to explore here would be, instead of model checking against the formulas mentioned above, to simply obtain the symbolic tableau NuSMV internally builds for $\Phi$ and compare this with the hypothesis automaton to examine whether behavior appearing in the former is missing from the latter and vice versa. It is not immediately obvious how this comparison would be performed, as the two representations are quite different in nature; nevertheless, this is something worth exploring, as it could potentially increase the efficiency of equivalence queries.

### 4.4 Properties & Complexity

**Minimality** The proposed algorithm (i) returns a minimal (w.r.t. number of states) SDFA and (ii) guarantees that all intermediate SDFA hypotheses contain

strictly fewer states than the returned result. Both (i) and (ii) follow directly from the properties of the $L^*$ algorithm: The initial hypothesis contains a single state, and the number of states of subsequent hypotheses is monotonically increasing until it reaches the value corresponding to the minimal automaton.

**Membership Query Complexity** Based on the definitions for `eval` in 4.1, in order to compute `eval` on a node in the formula tree of the form $\phi \wedge \psi$, $\phi \vee \psi$, $\neg\phi$, $X\phi$, each of its children needs to be considered at most once. Similarly, in order to compute `eval` on a node of the form $G\phi$ or $\phi W \psi$, each of its children needs to be considered at most $n$ times, where $n$ is the trace length. It is easy to see that with arbitrary nesting of operators in the safety subset of LTL, each node in the tree of the formula will need to be examined at most $n^{m+1}$ times, where $m$ is the total number of $G$ and $W$ operators in the formula. Therefore, the complexity of a single membership query is polynomial on the trace length and exponential on the formula length.

**Equivalence Query Complexity** A bound for performing an equivalence query can be given by a bound on translating the safety LTL formula into CTL with fairness constraints and a bound on symbolic model checking of the hypothesis automaton. As the hypothesis automaton can reach a number of states doubly exponential on the length of the safety LTL formula, and the model checking step is linear on the size of the automaton, the worst-case complexity of equivalence queries is at least doubly exponential on the length of the formula to be translated. This result motivated the search for a modified approach that eliminates equivalence queries altogether, which we present in the following.

### 4.5 A Priori Information & Inductive Inference

The observation that equivalence queries are needed in order to discover states of the target automaton raises an interesting question: What if we have some sort of state information beforehand? This could be the actual states represented as words or something else like distinguishing suffixes collectively allowing to differentiate all states of the target automaton. The former would not be enough; if we simply put these words in *RED* we would violate an important invariant of the algorithm: Since *SFX* would only contain the empty word, potentially many of the words in *RED* would be *SFX*-equivalent. Therefore, distinguishing suffixes would need to also be put in *SFX*. As it turns out, the latter alone is enough, since, as long as the required distinguishing suffixes are present, all states of the target automaton will be discovered and put in *RED* in the process of filling the observation table, updating *BLUE* and promoting states from *BLUE* to *RED* as needed, without the need for any equivalence queries.

In the rest of this section we discuss how we could obtain such information for two counter property families (parametrized properties that express some sort of counting by means of repetition / nesting of $X$ operators) taken from aerospace domain use-cases, shown in Table 1.

Table 1: Counter family formulas

| N | Counter family A | Counter family B |
|---|---|---|
| 1 | $G(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr))$ | $G(\neg p \vee X(\neg q \vee r))$ |
| 2 | $G(\neg p \vee X(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr)))$ | $G(\neg p \vee X(\neg q \vee (r \wedge Xr)))$ |
| 3 | $G(\neg p \vee X(\neg p \vee X(\neg p \vee X(\neg p \vee \neg q \vee r \vee Xr))))$ | $G(\neg p \vee X(\neg q \vee (r \wedge X(r \wedge Xr))))$ |

Suppose that we need to translate a formula of one of the families above for N = 50. What we could do is first translate formulas corresponding to small values of N, which would be fast since the formula size is small, then obtain the corresponding distinguishing suffixes (this can be easily done with breadth-first search), and finally employ an inductive inference procedure to identify a relation between N and the set of distinguishing suffixes, which, in turn, can be used to derive the required information for N = 50. Providing such an inductive inference procedure and formally analyzing its properties is outside the scope of this paper. However, for completeness, we outline a simple approach, generic enough to work on the property families listed above:

1. Identify base cases to be excluded from the following steps.
2. Identify how many suffixes are introduced from SDFA(N - 1) to SDFA(N)
3. For each newly introduced suffix in SDFA(N), identify a function to construct it from suffixes in SDFA(N - 1).

We first show example runs of the procedure outlined above on the two property families and then will explain how each individual step can be implemented in a general way. The relation between N and sets of distinguishing suffixes is shown in Table 2.

Table 2: Counter family formula suffixes

| N | Suffixes for counter family A | Suffixes for counter family B |
|---|---|---|
| 1 | $\{a,\ ba\}$ | $\{a\}$ |
| 2 | $\{a,\ ba,\ cba\}$ | $\{a,\ b,\ ca\}$ |
| 3 | $\{a,\ ba,\ cba,\ ccba\}$ | $\{a,\ b,\ ca,\ da,\ cca\}$ |
| 4 | $\{a,\ ba,\ cba,\ ccba,\ cccba\}$ | $\{a,\ b,\ ca,\ da,\ cca,\ dca,\ ccca\}$ |
| 5 | $\{a,\ ba,\ cba,\ ccba,\ cccba,\ ccccba\}$ | $\{a,\ b,\ ca,\ da,\ cca,\ dca,\ ccca,\ dcca,\ cccca\}$ |

where, for family A, we have $a := \neg p \wedge q \wedge \neg r$, $b := p \wedge q \wedge \neg r$ and $c := p \wedge q \wedge r$. We treat N = 1 as the base case, observe that when moving from N - 1 to N one new suffix is added, and notice that this suffix can be constructed by prepending $c$ to the longest suffix from step N - 1. For family B, we have $a := p \wedge q \wedge \neg r$, $b := p \wedge \neg q \wedge \neg r$, $c := \neg p \wedge q \wedge r$ and $d := \neg p \wedge \neg q \wedge r$. In this case, we treat N = 1 and N = 2 as base cases, and observe that two new suffixes are added when moving from N - 1 to N, which can be generated by taking the longest suffix from step N - 1 and (i) replacing the front letter with $d$ for one, and (ii) prepending the letter $c$ for the other.

Generalizing the above approach, step 1 (identifying base cases) can be performed by always treating N = 1 as a base case and then adding those cases the suffixes of which contain less distinct letters than the following cases, while

steps 2 and 3 can easily be formulated as *Syntax Guided Synthesis* [5] problems and solved as such.

## 5 Experimental Evaluation

We implemented a prototype version of the proposed algorithm (we refer to this as 'Proposed' throughout this Section) in the programming language D [1] and compared against scheck v1.2 [21], Spot v2.6.1 [3] and Rabinizer v4 [19] [4] on (i) 500 randomly generated syntactically safe LTL formulas, (ii) 54 formulas from the Spot benchmarks [2], as well as (iii) the 2 counter formula families from Section 4.5 and their conjunction. The 500 random formulas were generated using the Spot command line tools, and specifically the command:

```
randltl -n -1 4 --tree-size=10..30 -r | \
ltlfilt --lbt --syntactic-safety --size=10..25 -u -n 500
```

In short, the above means that we want 500 unique syntactic safety formulas with up to 4 atomic propositions, of length between 10 and 25. The 54 Spot benchmark formulas were taken from [2] and were the result of filtering the 184 formulas in that page for syntactic safety. The 2 counter formula families come from industrial (United Technologies Research Centre) requirements for aerospace domain digital hardware verification[5]. All experiments were run on an Ubuntu 14.04 laptop with a 1.6 GHz Intel Celeron processor and 4 GB of RAM. The results are summarized in Table 3 and Figures 2 and 3 (memory consumption generally closely follows running time in all cases).

Table 3: Execution times (in seconds) for 500 random and 54 Spot formulas

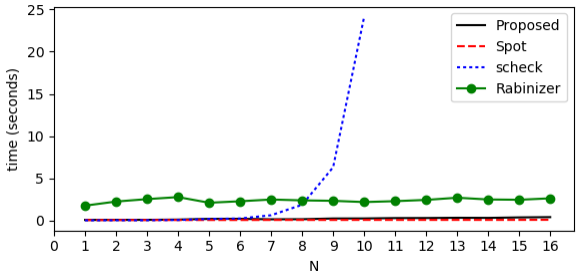
| Algorithm | 500 random formulas | | 54 Spot formulas | |
|---|---|---|---|---|
| | Average | Median | Average | Median |
| Proposed | 0.0693 | 0.0457 | 0.1262 | 0.0545 |
| Spot | 0.0397 | 0.0373 | 0.0406 | 0.0401 |
| scheck | 0.0082 | 0.0065 | 0.0161 | 0.0072 |
| Rabinizer | 1.4821 | 1.3668 | 1.8128 | 1.6885 |

As can be seen in Table 3, the proposed approach behaves comparably to others on small formulas. We argue that there is potential for improvement here by addressing some implementation details: In our prototype implementation, communication with NuSMV involves a lot of process and file I/O, which can cause considerable overhead (this is especially true for the latter, since hard disk

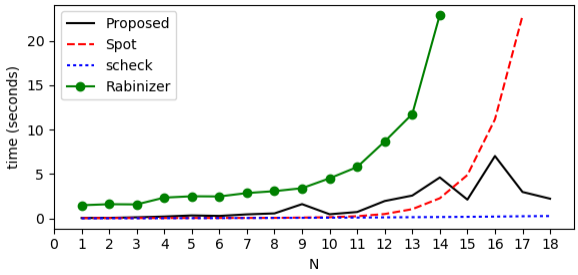
[4] To be fair to Rabinizer, since it is implemented in Java, we deducted 0.4 seconds (the measured JVM startup time) from the elapsed time in all experiments with it.

[5] Note that formulas of this kind with many (typically $> 50$) nested next operators, expressing timing requirements for FPGAs, appear very frequently in this domain.

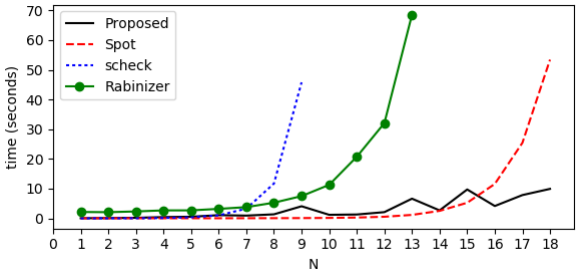
access is orders of magnitude slower than RAM access). In addition to that, the current implementation of equivalence queries does not take advantage of the fact that parts of past SDFA hypotheses exist in future ones – an incremental model checking approach would be of great benefit here. This last issue, in particular, is responsible for some spikes in running time that drive the average away from the median in our case. Regarding the size of the corresponding minimal automata, the average and median number of states are 5.4 and 5 in the random formulas case, and 4.1 and 4 for the Spot formulas. Note that, in all cases, all tools except Rabinizer return a minimal automaton.




(a) Counter family A



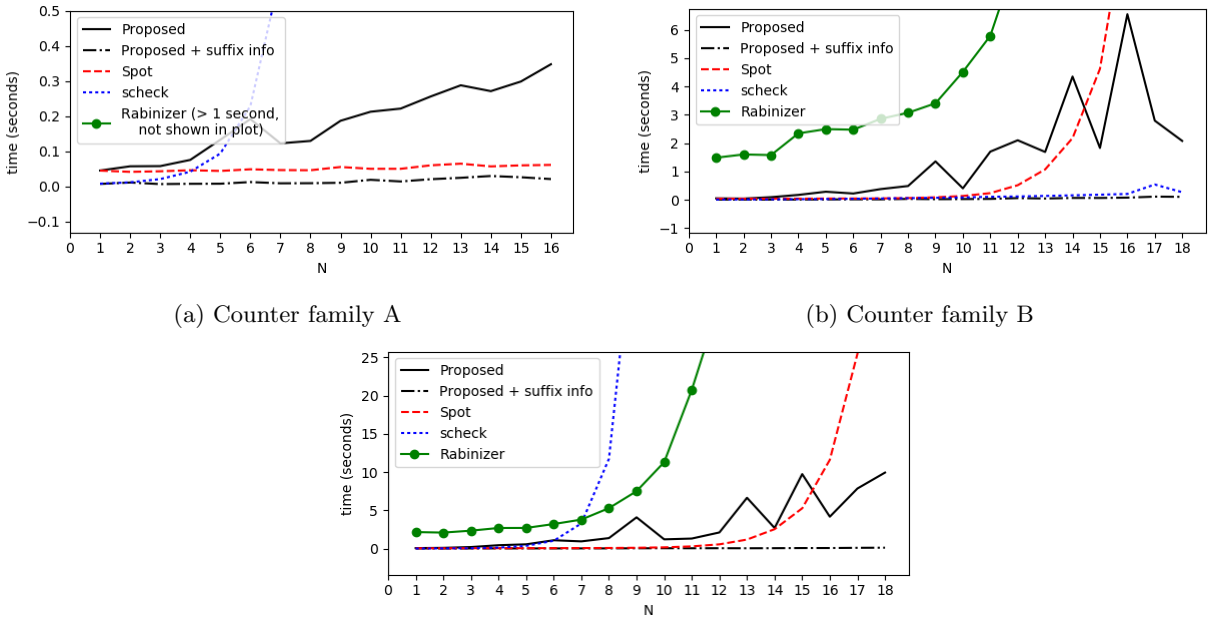

(b) Counter family B




(c) Counter family conjunction

Fig. 2: Results on counter formulas

Where our learning approach shines is in translating the longer counter formulas (Figure 2). It performs better asymptotically, as the number of next operators increases. Even if the automata grow linearly in size with the number of X operators ($N + 3$ states for counter family A and $2N + 1$ states for counter family B and their conjunction), scheck, Spot and Rabinizer require exponential time in at least one of the property families and in their conjunction, while our approach requires only linear time in all cases. This is a direct manifestation of the main drawback of conventional translation approaches; we remind here that our approach provides theoretical guarantees that this intermediate result explosion does not happen. Also note that for counter family B as well as for




(a) Counter family A

(b) Counter family B

(c) Counter family conjunction

Fig. 3: Effect of suffix information on counter formulas

the conjunction of counter family formulas Rabinizer does not return minimal automata (it returns automata that grow exponentially in size with N).

Taking into account a priori information about the target automaton gives a significant additional boost to the proposed approach, as shown in Figure 3. Note that, while not very clear in the figures, the proposed approach using suffix information performs better than all other tools on both counter formulas (and their conjunction) for all values of N. These graphs also provide an idea of the overhead introduced by the current implementation of equivalence queries due to the non-incremental model checking approach in NuSMV as well as communication delays (process and file I/O).

## 6 Conclusion & Future Work

In this work we presented a learning-based approach for translating safety LTL to DFA. We studied its theoretical properties and demonstrated its perfomance in practice. The proposed approach is comparable with existing ones in formulas of small size. Moreover, by guaranteeing that intermediate results do not explode in size, it outperforms existing approaches in long instances of important property families, by orders of magnitude. In addition, unlike existing approaches, it can take into account a priori information about the target automaton, which leads to even better performance.

We believe that the proposed approach nicely complements existing LTL translators in the sense that, performance-wise, a hybrid approach where (i) existing translators are used for small formulas and for inductive inference of the suffix information, and (ii) the proposed approach with the previously derived suffix information is used for longer formulas, would behave best.

In the future, we plan to improve our learning-based approach by employing more $L^*$ optimizations (e.g. parallel membership queries, $TTT$ algorithm [18]), and by using an incremental model checking approach for equivalence queries. We also plan to extend this work to translate general (not just safety) LTL properties to Büchi automata as well [7, 23].

## References


1. D programming language. `https://dlang.org/`.
2. Spot 1.0 benchmarks. `https://www.lrde.epita.fr/~adl/ijccbs/`.
3. Alexandre Duret-Lutz and Alexandre Lewkowicz and Amaury Fauchille and Thibaud Michaud and Etienne Renault and Laurent Xu. Spot 2.0 — a framework for LTL and $\omega$-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, oct 2016.
4. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, Sep 1987.
5. R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, Oct 2013.
6. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.
7. D. Angluin and D. Fisman. Learning Regular Omega Languages. In P. Auer, A. Clark, T. Zeugmann, and S. Zilles, editors, *Algorithmic Learning Theory*, pages 125–139, Cham, 2014. Springer International Publishing.
8. T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. *CoRR*, abs/1201.0682, 2012.
9. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
10. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, Feb 1997.
11. L. D'Antoni and M. Veanes. The Power of Symbolic Automata and Transducers. In *CAV*, 2017.
12. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.
13. S. Drews and L. D'Antoni. Learning Symbolic Automata. In *TACAS*, 2017.
14. E. A. Emerson and C.-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract). In *LICS*, 1986.

15. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
16. M. Geilen. On the Construction of Monitors for Temporal Logic Properties. *Electronic Notes in Theoretical Computer Science*, 55(2):181 – 199, 2001. RV'2001, Runtime Verification (in connection with CAV '01).
17. F. Howar, B. Steffen, and M. Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 263–277, Berlin, Heidelberg, 2011. Springer-Verlag.
18. M. Isberner, F. Howar, and B. Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.
19. J. Křetínský, T. Meggendorfer, S. Sickert, and C. Ziegler. Rabinizer 4: From LTL to Your Favourite Deterministic Automaton. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 567–577, Cham, 2018. Springer International Publishing.
20. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, Nov 2001.
21. T. Latvala. Efficient Model Checking of Safety Properties. In T. Ball and S. K. Rajamani, editors, *Model Checking Software*, pages 74–88, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
22. O. Maler and I.-E. Mens. Learning Regular Languages over Large Alphabets. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 485–499, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
23. O. Maler and A. Pnueli. On the Learnability of Infinitary Regular Sets. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, COLT '91, pages 128–138, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
24. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
25. K. Y. Rozier. Survey: Linear Temporal Logic Symbolic Model Checking. *Comput. Sci. Rev.*, 5(2):163–203, May 2011.
26. R. Sebastiani and S. Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods*, pages 126–140, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
27. M. Shahbaz and R. Groz. Inferring Mealy Machines. In A. Cavalcanti and D. R. Dams, editors, *FM 2009: Formal Methods*, pages 207–222, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
28. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, Sep 1994.