Verifying the LTL to Büchi Automata Translation via Very Weak Alternating Automata

Simon Jantsch[1] and Michael Norrish[2]

[1] TU Dresden
[2] Data61, CSIRO and Australian National University

Abstract. We present a formalization of a translation from LTL formulae to generalized Büchi automata in the HOL4 theorem prover. Translations from temporal logics to automata are at the core of model checking algorithms based on automata-theoretic techniques. The translation we verify proceeds in two steps: it produces very weak alternating automata at an intermediate stage, and then ultimately produces a generalized Büchi automaton. After verifying both transformations, we also encode both of these automata models using a generic, functional graph type, and use the CakeML compiler to generate fully verified machine code implementing the translation.

1 Introduction

As the goal of verification techniques is to give the user of a system guarantees about its behaviour, bugs in verification tools can potentially have severe consequences and considerably reduce the trust of users in the techniques. While new verification algorithms are usually proven correct on paper, the gap between the abstract proof and any actual implementation can be large. Many times different representations are used and optimizations are added that are not considered in the proofs.

Our aim is to bridge this gap for one standard algorithm used for automata-based LTL model checking. The algorithm, by Gastin and Oddoux [7] (G&O henceforth), improves on the efficiency of the translation of LTL formulae into automata. Rather than moving directly from such formulae into generalized Büchi automata (GBA), it introduces an intermediate step, the rather complicated *alternating automata*. Whereas the efficient translation from LTL to alternating automata was known before, G&O showed that a property, namely very weakness, of the resulting automata can be exploited for the translation to GBA.

This new step represents an advantage on earlier techniques in part because automata-optimizations can be applied in both phases. Optimizing the alternating automaton is especially interesting as it is linear in the size of the formula,

[1] The author was supported by the European Master's Program in Computational Logic (EMCL).

even though the final GBA may still be of exponential size. As noted in Schimpf *et al.* [15], the original tool implementing this algorithm contained a bug that went unnoticed for several years despite widespread use.

Translations of LTL formulae to automata play a core role in LTL model checking. In the usual approach, an LTL formula φ is given together with a labeled transition system $\mathcal{S}$, and the questions is whether all executions of $\mathcal{S}$ satisfy the formula φ. To check this, an automaton is constructed for $\neg\varphi$, which is then combined with an automaton describing all executions of $\mathcal{S}$. If the combined automaton is empty, $\mathcal{S}$ indeed satisfies the property specified by φ, otherwise a counterexample to this claim can be given.

To obtain a formally verified implementation of the algorithm in G&O, we proceed as follows: first we formalize the procedure in an abstract way, using set notation and mathematical functions. We prove correctness of this function, which is a mechanization of the proof given in G&O. Then we implement another version of the algorithm, now defined on concrete data structures that represent the automata in a compact way. In contrast to the first function, this second version describes an algorithm: a step-by-step expansion of a graph.

The relation between our two versions is established by defining abstraction functions from our concrete automata to their abstract counterparts. Using these functions, we show that the automata we obtain in our concrete algorithm coincide with the abstract automata, for which we have proved the desired property. One strength of this approach is that it lets us separate the correctness proofs of the main function and the restriction to reachable states on the abstract level, while still combining the two functions on the concrete level in a single expansion algorithm. We believe that this idea can be extended to add optimizations to the translation in a manageable way by defining them as seperate transformation steps on the abstract level, and efficiently embedding them into the expansion algorithm on the concrete level.

Finally, we compile our function into machine code using the CakeML compiler. This adds another guarantee to our implementation, as we do not have to trust the translation of the algorithm as expressed in HOL4 into SML, nor the correctness of an SML compiler. The proof scripts and definitions for our translation are available as part of the HOL4 system, and the scripts to compile the algorithm with CakeML are available on `Gitlab`.[3]

The paper is structured as follows: Section 2 introduces LTL and the automata models we consider. Section 3 recalls the algorithm in G&O, and Section 4 discusses our formalization in HOL4. Section 5 gives an overview of related work, and we conclude in Section 6.

[3] For the abstract and concrete algorithms, see the `examples/logic/ltl` directory in HOL4 after commit `b4576ed`, and see `https://gitlab.com/simon-jantsch/ltl2baHol-paper/tree/master/cmlltl` for our CakeML translations, which in turn depend on CakeML commit `891cbf4a`.

2 Preliminaries

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a logic that extends propositional logic with temporal operators. We define it using unary $\mathbf{X}$ ("next") and binary $\mathbf{U}$ ("until").

Definition 1 (Syntax of LTL) *Given a set of atomic propositions AP, the set of LTL formulae over AP is defined with the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

where $p \in AP$

An interpretation of an LTL formula is a sequence of propositional valuations over AP, one for each point in time. This sequence is viewed as an infinite word over $\mathscr{P}(AP)$ (we write $\mathscr{P}(S)$ to mean the powerset of S). The symbol of w at position i is denoted by $w[i]$ and the suffix of w starting at position i by $w[i..]$. Given $w \in (\mathscr{P}(AP))^\omega$, we define

Definition 2 (Semantics of LTL)

$$\begin{array}{ll}
w \models p & \textit{iff } p \in w[0], \textit{ for all } p \in AP \\
w \models \neg\varphi & \textit{iff } w \not\models \varphi \\
w \models \varphi_1 \wedge \varphi_2 & \textit{iff } w \models \varphi_1 \textit{ and } w \models \varphi_2 \\
w \models \mathbf{X}\varphi & \textit{iff } w[1..] \models \varphi \\
w \models \varphi_1\mathbf{U}\varphi_2 & \textit{iff } \exists i.\ w[i..] \models \varphi_2 \textit{ and } \forall j < i.\ w[j..] \models \varphi_1
\end{array}$$

As we want to use a negation normal form we introduce the dual operators $\vee$ and $\varphi_1\mathbf{R}\varphi_2 = \neg(\neg\varphi_1\mathbf{U}\neg\varphi_2)$. An LTL formula φ is in negation normal form if all occurrences of $\neg$ are directly in front of an atomic proposition. We call a formula a *temporal formula* if it is a (possibly negated) atomic proposition or if its outermost operator is $\mathbf{X}$, $\mathbf{U}$ or $\mathbf{R}$. We use $\mathcal{L}(\varphi) = \{w \in (\mathscr{P}(AP))^\omega \mid w \models \varphi\}$ to denote the *language* of an LTL formula.

As the semantics of LTL is defined using infinite words, questions about LTL formulae can often be formulated as word problems. This is where automata, in our case recognizing languages of infinite words, come into play. In the following sections we introduce the two automata types used in G&O, beginning with alternating automata.

2.2 Co-Büchi alternating automata

In an *alternating automaton*, each state nondeterministically chooses between *sets* of successor states. Intuitively, a word $w = a_0a_1\ldots$ is accepted from a state q if there *exists* a successor set S reachable via the symbol a_0 such that $a_1a_2\ldots$ is accepted from *all* states in S.

Definition 3 *A co-Büchi alternating automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \to \mathscr{P}(\mathscr{P}(\Sigma) \times \mathscr{P}(Q))$, $I \subseteq \mathscr{P}(Q)$ is the set of initial sets and $F \subseteq Q$ is the set of final states.*

Alternating automata can be defined with different acceptance conditions but we will always mean co-Büchi alternating automata in what follows. In HOL4 we use the following datatype for abstract alternating automata:

```
(α, σ) ALTER_A = <|
  states : σ set;
  alphabet : α set;
  trans : σ → (α set × σ set) set;
  initial : σ set set;
  final : σ set
|>
```

The transition function δ assigns to each state in the automaton a set of pairs (A, S), where $A \subseteq \Sigma$ and $S \subseteq Q$. Such a pair stands for a transition that is active for every symbol in A and has successor set S. This definition of alternating automata was introduced in G&O and differs from the more usual definition, where the transition function is defined using positive boolean formulae over the states (e.g. Löding [11] or Vardi [17]). As noted in G&O, the two can easily be transformed into each other: the presented definition corresponds closely to the disjunctive normal form of the positive boolean formula.

Following Löding [11] we define a run of an alternating automaton $\mathcal{A}$ on a word $w \in \Sigma^\omega$ as a directed acyclic graph $\rho = (V, E)$, where $V \subseteq Q \times \mathbb{N}$, $E \subseteq \bigcup_{i \geq 0}(Q \times \{i\}) \times (Q \times \{i+1\})$ and

- $\{q \mid (q, 0) \in V\} \in I$;
- for all $(q, i) \in V$ there exists $(A, S) \in \delta(q)$ such that $w[i] \in A$ and $\{q' \mid ((q, i), (q', i+1)) \in E\} = S$; and
- for all $(q, i) \in V$ where $i > 0$, there exists some $(q_p, i-1) \in V$ such that $((q_p, i-1), (q, i)) \in E$

For co-Büchi automata, acceptance is defined as follows: a run ρ is accepting if there is no path through ρ that visits a state in F infinitely often. The language of a co-Büchi alternating automaton is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid$ there exists an accepting run of $\mathcal{A}$ on $w\}$.

Note that the transition function allows empty successor sets. Such a transition corresponds to the empty conjunction (i.e. *true*) and leads to direct acceptance of any suffix word for which it is active.

An alternating automaton is *very weak* if there is a partial order R on Q, such that whenever $(A, S) \in \delta(q)$ and $q' \in S$ then $R\,q'\,q$. As Q is finite, this implies that all loops in the automaton are self-loops and every path in a run ρ ultimately stabilizes on some state.

2.3 Generalized Büchi automata

The algorithm we consider produces generalized Büchi automata (GBA), where the acceptance condition is defined using the edges, rather than the states, of the automaton.

Definition 4 *A generalized Büchi automaton is a tuple $\mathcal{G} = (Q, \Sigma, \delta, I, T)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \to \mathcal{P}(\mathcal{P}(\Sigma) \times Q)$ is the transition function, $I \subseteq Q$ is the set of initial states and $\mathcal{T} = \{T_1, T_2, \ldots\}$, with $T_i \subseteq Q \times \mathcal{P}(\Sigma) \times Q$, is a set of sets of accepting edges.*

A run $r = q_0 q_1 \ldots \in Q^\omega$ of a GBA $\mathcal{G}$ on a word $w \in \Sigma^\omega$ is a sequence of states such that $q_0 \in I$ and for all i there exists a pair $(A, q_{i+1}) \in \delta(q_i)$ such that $w[i] \in A$. It is accepting if for all $T \in \mathcal{T}$ there exist infinitely many positions i such that for some A: $(A, q_{i+1}) \in \delta(q_i)$, $w[i] \in A$ and $(q_i, A, q_{i+1}) \in T$. The language of a GBA is defined accordingly: $\mathcal{L}(\mathcal{G}) = \{w \in \Sigma^\omega \mid \text{there exists an accepting run of } \mathcal{G} \text{ on } w\}$.

GBA can be transformed into ordinary Büchi automata via a standard linear transformation called degeneralization. The emptiness check, which is required for LTL model checking, can be done on Büchi automata efficiently [2]. However, approaches have been developed to use the GBA directly to check emptiness, thereby omitting degeneralization [3].

3 Translating LTL to GBA

We now recall the translation presented in G&O. The algorithm proceeds in two steps: it first translates an LTL formula into an equivalent very weak alternating automaton (VWAA), and then translates that VWAA into a GBA. By "equivalent", we mean that the words accepted by the VWAA are exactly the words that satisfy the formula, as per Definition 2.

We introduce two functions that we need for the definition, $\overline{\varphi}$ gives an approximation of the DNF of φ without simplifying temporal subformulae. $\otimes$ is an operation on the transitions of the VWAA that corresponds to conjunction on the formula level. From now on we consider all formulae to be in negation normal form.

Definition 5 *Let φ be an LTL formula. $\overline{\varphi} = \{\{\varphi\}\}$ if φ is a temporal formula, $\overline{\varphi \wedge \psi} = \{S_1 \cup S_2 \mid S_1 \in \overline{\varphi} \text{ and } S_2 \in \overline{\psi}\}$ and $\overline{\varphi \vee \psi} = \overline{\varphi} \cup \overline{\psi}$.*

Let $D_1, D_2 \in \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(Q)$. $D_1 \otimes D_2 = \{(A_1 \cap A_2, S_1 \cup S_2) \mid (A_1, S_1) \in D_1$ and $(A_2, S_2) \in D_2\}$.

Now we can define the first step of the translation. It models the boolean structure of the formulae with the transitions of the VWAA and makes use of the equalities $\varphi \mathbf{U} \psi = \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi))$ and $\varphi \mathbf{R} \psi = \psi \wedge (\varphi \vee \mathbf{X}(\varphi \mathbf{R} \psi))$.

Definition 6 *Let φ be an LTL formula over AP. We define $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, F)$, where Q is the set of temporal subformulae of φ, $\Sigma = \mathscr{P}(AP)$, $I = \overline{\varphi}$, F is the set of subformulae of φ of the type $\psi_1 \mathbf{U} \psi_2$ and δ is defined by:*

$$\begin{aligned}
\delta(p) &= \{(\Sigma_p, \emptyset)\},\ \textit{where } \Sigma_p = \{A \in \Sigma \mid p \in A\} \\
\delta(\neg p) &= \{(\Sigma_{\neg p}, \emptyset)\},\ \textit{where } \Sigma_{\neg p} = \Sigma \setminus \Sigma_p \\
\delta(\mathbf{X}\psi) &= \{(\Sigma, S) \mid S \in \overline{\psi}\} \\
\delta(\psi_1 \mathbf{U} \psi_2) &= \Delta(\psi_2) \cup (\Delta(\psi_1) \otimes \{(\Sigma, \{\psi_1 \mathbf{U} \psi_2\})\}) \\
\delta(\psi_1 \mathbf{R} \psi_2) &= \Delta(\psi_2) \otimes (\Delta(\psi_1) \cup \{(\Sigma, \{\psi_1 \mathbf{R} \psi_2\})\}) \\
\Delta(\psi) &= \delta(\psi),\ \textit{if } \psi \textit{ is a temporal formula} \\
\Delta(\psi_1 \wedge \psi_2) &= \Delta(\psi_1) \otimes \Delta(\psi_2) \\
\Delta(\psi_1 \vee \psi_2) &= \Delta(\psi_1) \cup \Delta(\psi_2)
\end{aligned}$$

As every transition contains only subformulae of the considered formula, we see that $\mathcal{A}_\varphi$ is very weak. In G&O the following theorem is stated without a complete proof. We discuss our proof and its mechanization in Section 4. A proof for the standard setting, which simplifies the proof, can be found in Vardi [17].

Theorem 1 $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$

The second step of the algorithm is a translation of a VWAA into a GBA. To define it we will use the following relation on the transitions of the resulting automaton: Let $t_1 = (S, A_1, S'_1)$ and $t_2 = (S, A_2, S'_2)$ be transitions of the GBA. Then $t_1 \preceq t_2$ if $A_2 \subseteq A_1$, $S'_1 \subseteq S'_2$ and for all $T \in \mathcal{T}$: $t_2 \in T \Rightarrow t_1 \in T$. We can now define the translation from VWAA to GBA.

Definition 7 *Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a VWAA. We define $\mathcal{G}_\mathcal{A} = (\mathscr{P}(Q), \Sigma, \delta', I, \mathcal{T})$, where*

- *$\delta'(\{q_0, q_1, \ldots, q_n\})$ is the set of $\preceq$-minimal transitions in $\bigotimes_{i=0}^{n} \delta(q_i)$*
- *$\mathcal{T} = \{T_f \mid f \in F\}$, where $T_f = \{(S, A, S') \mid f \notin S'$ or there is $(B, X) \in \delta(f)$ such that $A \subseteq B$ and $f \notin X \subseteq S'\}$*

An example of the translations to VWAA and GBA is given in Figure 1.

Theorem 2 $\mathcal{L}(\mathcal{G}_\mathcal{A}) = \mathcal{L}(\mathcal{A})$

Proof. See G&O for a proof.

4 Verifying the algorithm

Note that the way the translation is presented is far from an actual implementation. In particular the worst case complexity is always exhibited as nonreachable

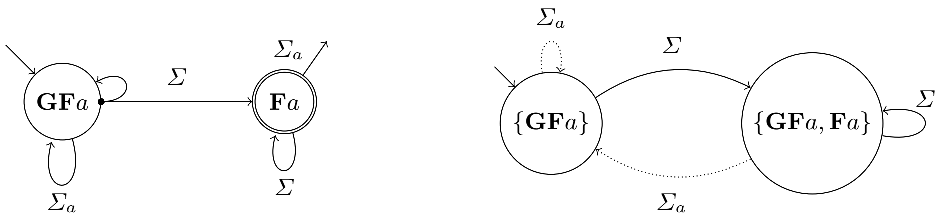

Fig. 1: Translation of the formula $\mathbf{GF}a$ into a VWAA (left) and a GBA (right). Here $\mathbf{F}\,\varphi$ (eventually) and $\mathbf{G}\,\varphi$ (always) abbreviate $\mathit{true}\mathbf{U}\varphi$ and $\neg\mathbf{F}\neg\varphi$ respectively. Transitions conjoined with • are conjunctive transitions to multiple successors. Recall that Σ_a is the set of all elements in Σ that contain a. Arrows with no successor node indicate transitions to the empty set. Final states in the VWAA are indicated by doubled circles and accepting transitions (of the single acceptance set) in the GBA are indicated by a dotted line.

states are not excluded. Also the way the transitions are defined, where the first component is a set of subsets of AP, is prohibitively inefficient. These representations are convenient for the proofs, but the question is how exactly any concrete algorithm relates to this abstract description. We introduce a more compact representation and define its relation to the abstract one.

Figure 2 visualizes our approach. As in G&O, we do not worry about reachable states in our main correctness proof; rather we implement the restriction on the reachable states as a separate function (restr_states in Figure 2).

$$\begin{tikzcd}
 & \varphi \arrow[dl, Rightarrow, "\mathsf{concr\_ltl2vwaa}"'] \arrow[dr, "\mathsf{ltl2vwaa}"] & \\
\textbf{concrAA} \arrow[drr, dotted, "\mathsf{abstr\_AA}"'] & & \mathcal{A} \arrow[d, "\mathsf{restr\_states}"] \\
 & & \mathcal{A}_r
\end{tikzcd}
\qquad
\begin{tikzcd}
\textbf{concrAA} \arrow[d, Rightarrow, "\mathsf{concr\_vwaa2gba}"] & \mathcal{A}_r \arrow[d, "\mathsf{vwaa2gba}"] \\
\textbf{concrGBA} \arrow[dr, dotted, "\mathsf{abstr\_GBA}"'] & \mathcal{G}_{\mathcal{A}_r} \arrow[d, "\mathsf{restr\_states}"] \\
 & \mathcal{G}_{r\mathcal{A}_r}
\end{tikzcd}$$

Fig. 2: Dividing the formalization into abstract and concrete parts. Thick arrows represent concrete functions, thin arrows represent abstract functions, and dotted arrows are abstractions from concrete to abstract automata. An r in the subscript stands for a restriction to reachable states.

4.1 Mechanizing the abstract proofs

Our abstract formalizations in HOL4 are basically identical to the mathematical definitions given in Section 2. This allows us to closely follow the proof of

Theorem 2 from G&O. First, we discuss the proof of Theorem 1, which is not presented in G&O:

$$\vdash \mathcal{L}\ \phi \ = \ \mathcal{L}_{AA}\ (\mathsf{ltl2vwaa}\ \phi)$$

In the proof we fix a formula ϕ and with it the alphabet we are considering, namely $\mathscr{P}(\mathsf{props}\ \phi)$, where props is the function that collects all atomic propositions that occur in a formula. Then we show the claim for all subformulae of ϕ by structural induction on LTL formulae.

The base case is the translation of an atomic proposition $p \in \mathsf{props}\ \phi$. The corresponding automaton $\mathcal{A}_p$ has one state with transitions to the empty set for all elements in $\mathscr{P}(\mathsf{props}\ \phi)$ that contain p. Thus the automaton accepts exactly the words w for which such a transition is active, which is the case exactly if $p \in w[0]$.

In the other cases, we show how accepting runs of the sub-automata can be used to build accepting runs of the automata of the current case. Consider the case $\mathbf{X}\psi$. For any word w such that $w \models \mathbf{X}\psi$ we get $w[1..] \models \psi$ and by induction hypothesis an accepting run of $\mathcal{A}_\psi$ on $w[1..]$. By shifting this run by one and adding the vertex $(\mathbf{X}\psi, 0)$, we get an accepting run of $\mathcal{A}_{\mathbf{X}\psi}$. For the other direction we start with an accepting run of $\mathcal{A}_{\mathbf{X}\psi}$ on w. By the structure of $\mathcal{A}_{\mathbf{X}\psi}$ we can extract a run of $\mathcal{A}_\psi$ on the word $w[1..]$. This is done by again shifting the run by one, but now in the other direction. Applying the induction hypothesis yields $w[1..] \models \psi$, from which we can conclude $w \models \mathbf{X}\psi$.

The existence of these two runs is shown in the proofs for the following lemmata:[4]

$$\vdash \mathsf{runOfAA}\ (\mathsf{ltl2vwaa}_\phi\ \psi)\ r\ w[1..] \ \wedge\ \mathsf{word\_range}\ w\ \subseteq\ \mathscr{P}(\mathsf{props}\ \phi)\ \Rightarrow \\ \exists\, r'.\ \mathsf{runOfAA}\ (\mathsf{ltl2vwaa}_\phi\ (\mathsf{X}\ \psi))\ r'\ w$$

$$\vdash \mathsf{runOfAA}\ (\mathsf{ltl2vwaa}_\phi\ (\mathsf{X}\ \psi))\ r\ w\ \Rightarrow\ \exists\, r'.\ \mathsf{runOfAA}\ (\mathsf{ltl2vwaa}_\phi\ \psi)\ r'\ w[1..]$$

The expression $\mathsf{ltl2vwaa}_\phi\ \psi$ denotes the automaton for ψ, as defined by Definition 6, with respect to the alphabet $\mathscr{P}(\mathsf{props}\ \phi)$. (In particular, $\mathsf{ltl2vwaa}\ \phi = \mathsf{ltl2vwaa}_\phi\ \phi$.) The condition runOfAA *aut* r w states that r is a run of *aut* on w.

To show acceptance of the runs we construct in this case we use the fact that the final states of the automata $\mathcal{A}_\psi$ and $\mathcal{A}_{\mathbf{X}\psi}$ are the same, as no "until"-formula is added to the automaton in the $\mathbf{X}$ case. So it is enough if we can map every path in the run we construct to some path in the old run that visits the same set of nodes infinitely often. This is clearly possible as the only way we transformed the runs was to shift them by one.

The most interesting cases are the temporal operators $\mathbf{U}$ and $\mathbf{R}$, where the acceptance conditions become important. In $\varphi\mathbf{U}\psi$, for example, we first show that the automaton cannot stay in the state $\varphi\mathbf{U}\psi$ forever, as this would lead to a rejecting path in the corresponding run. This is because all "until"-formulae

[4] We need the precondition $\mathsf{word\_range}\ w\ \subseteq\ \mathscr{P}(\mathsf{props}\ \phi)$ to make sure that w is a word over the alphabet $\mathscr{P}(\mathsf{props}\ \phi)$ as we have no restriction on $w[0]$ otherwise.

are final states in our automata, and the co-Büchi condition requires an accepting run to have no paths visiting infinitely many final states. At the position where $\varphi\mathbf{U}\psi$ no longer loops, its next transition needs to be a transition of ψ, by Definition 6. Thus we can extract an accepting run of $\mathcal{A}_\psi$ for the suffix word starting at that position. For all positions until that point we can extract runs of $\mathcal{A}_\varphi$ and thus, via induction hypothesis, show that the word satisfies $\varphi\mathbf{U}\psi$.

The correctness of the second part, from VWAA to GBA, is captured in the following theorem.

$$\begin{array}{l}\vdash \mathsf{isVeryWeakAA}\ a_{AA}\ \wedge\ \mathsf{FINITE}\ a_{AA}.\mathsf{alphabet}\ \wedge\ \mathsf{FINITE}\ a_{AA}.\mathsf{states}\ \wedge \\ \quad \mathsf{isValidAA}\ a_{AA}\ \Rightarrow \\ \qquad \mathcal{L}_{GB}\ (\mathsf{vwaa2gba}\ a_{AA})\ =\ \mathcal{L}_{AA}\ a_{AA}\end{array}$$

We have to show that for every accepting run of the VWAA on a word w, there exists an accepting run of the GBA on w, and vice versa. By the way the GBA transitions are defined it can be seen that the sequence of layers in a run of the VWAA corresponds to a run of the GBA. The two main difficulties are to cope with the reduction of transitions by $\preceq$ in Definition 7 and to show acceptance of the runs. As our formalization follows the proof in G&O closely, we omit the details here.

Finally we show that we can restrict our automata to reachable states, by proving that no state that is not reachable can appear in any run of the corresponding automaton. We define a function for each automata model, with the overloaded name restr_states, that implements this restriction.

$$\vdash \mathcal{L}_{AA}\ a_{AA}\ =\ \mathcal{L}_{AA}\ (\mathsf{restr\_states}\ a_{AA})$$

$$\vdash \mathsf{isValidGBA}\ a_{GB}\ \Rightarrow\ \mathcal{L}_{GB}\ a_{GB}\ =\ \mathcal{L}_{GB}\ (\mathsf{restr\_states}\ a_{GB})$$

4.2 Concrete data structures

We use the following generic finite graph type to implement concrete representations of our automata in HOL4:

```
(α, ϵ) gfg = <|
  node_info : α spt;
  followers : (ϵ × num) list spt;
  preds : (ϵ × num) list spt;
  next : num
|>
```

The α *spt* type implements a dictionary with keys that are natural numbers and values of type α. Thus, a graph contains a set of nodes uniquely labeled with natural numbers. Each node is associated with "node information" (the α type parameter). In addition, dictionaries map each node label to outgoing and incoming edges, where each edge connects to another node (identified by the *num*), and "edge information" (the ϵ parameter). Finally, the next field tracks the next node label, to be used when a node is inserted. This representation is inspired by Erwig [4], and is readily translated into CakeML.

The types used to capture node and edge information are given in Figure 3. As the transition structure of alternating automata allows conjunctive transitions to several successors we cannot directly map it into the transition structure of the graph. To solve this we extend the edge labels by a field called edge_grp. Multiple edges with the same value of edge_grp are meant to belong to the same conjunctive edge of the alternating automaton. The set of symbols of Σ for which the transition is active is represented using two lists of atomic propositions, one for positive and one for negative occurrences. This is possible because the first component of any transition is always the result of intersecting sets Σ,Σ_p and $\Sigma_{\neg p}$, by Definition 6, which was observed in G&O. This explains the type of our edge labels α *edge_labelAA* as defined in Figure 3.

```
α edge_labelAA = <| edge_grp : num; pos_lab : α list; neg_lab : α list |>
α node_labelAA =
  <| frml : α ltl_frml; is_final : bool; true_labels : α edge_labelAA list |>
α concrAA = <|
  graph : (α node_labelAA, α edge_labelAA) gfg;
  init : num list list;
  atomic_prop : α list
|>
```

Fig. 3: Encoding the concrete representation of alternating automata.

Another aspect of alternating automata that cannot be captured immediately in the graph are transitions to an empty set of successors. One way to handle them is to add a state representing *true* from which any suffix word is accepted. As we do not have this state in our abstract automata in general, this would break the direct correspondence of states in our abstract and concrete models. We encode this information in the node labels of our concrete structure (α *node_labelAA*). Any edge label that appears in the field true_labels corresponds to an edge with the empty successor set in our abstract model.

Using these two types we define our concrete alternating automata by combining the graph with a list of atomic propositions and an init field corresponding to the set of sets given by I in the abstract automaton.

As the GBA transition structure corresponds to an ordinary graph, we can define it in the natural way (see Figure 4). By Definition 7, the states of the GBA are sets of states of the VWAA, which are LTL formulae in our case, so we label the GBA states by lists of LTL formulae. The acc_set field is a list of formulae for which the edge is accepting. So rather than grouping all the accepting edges in a set T_f, every edge that is accepting for f should contain f in the field acc_set. Additionally the field all_acc_frmls declares all acceptance sets that exist in the GBA.

```
α edge_labelGBA = <| pos_lab : α list; neg_lab : α list; acc_set : α ltl_frml list |>
α node_labelGBA = <| frmls : α ltl_frml list |>
α concrGBA = <|
  graph : (α node_labelGBA, α edge_labelGBA) gfg;
  init : num list;
  all_acc_frmls : α ltl_frml list;
  atomic_prop : α list
|>
```

Fig. 4: The types used to encode the concrete representation of GBAs.

4.3 Abstraction functions

To establish the correspondence between our concrete and abstract automata we define abstraction functions that take a concrete automaton and return its abstract counterpart. These abstraction functions can be seen as defining the semantics of the concrete structure.

To abstract the states of the automaton we visit all nodes in the graph and read their labels. For the transitions we introduce the following function:

```
transform_label AP pos neg =
  FOLDR (λ a sofar. char (𝒫(AP)) a ∩ sofar)
    (FOLDR (λ a sofar. char_neg (𝒫(AP)) a ∩ sofar) (𝒫(AP))
      neg) pos
```

The functions char and char_neg are defined exactly as the sets Σ_p and $\Sigma_{\neg p}$ in Definition 6, where $\Sigma = \mathscr{P}(AP)$ in this case. The function transform_label defines how the fields pos_lab and neg_lab of the concrete edge labels should be interpreted. It computes all subsets of Σ that contain all atomic propositions in *pos* and do not contain any atomic proposition in *neg*.

Note that different values of *pos* and *neg* can lead to the same abstract interpretation by transform_label. One reason is that the order of the lists does not matter, the other is that whenever some atomic proposition appears in both lists, the value of transform_label is the empty set.

To abstract the transition function we have to compute a set of abstract transitions given a formula φ. We do this by finding the node labeled by φ in the graph, grouping its outgoing edges by the value of edge_grp, looking up all the identifiers of the successor states and computing the first components of the transitions using transform_label. If there is no such node in the graph, the function returns the empty set. We call this function abstr_transAA. The procedure for the abstract GBA follows the same idea but does not have to bother with conjunctive edges.

The final states are abstracted by collecting all states of the concrete VWAA that have is_final set to *true*. From the concrete GBA we get the sets T_f by collecting all transitions where the acc_set field in the edge label contains the formula f.

4.4 Concrete translations

Concrete LTL to VWAA First we describe our concrete algorithm for the first part of the translation, from LTL formulae to VWAA, now encoded with the concrete graph types described in Section 4.2. We reimplement the core functions $\overline{\varphi}$ and $\otimes$ and a concrete version of δ, called concr_trans, using lists, and show that when abstracted with transform_label, concr_trans corresponds to δ.

Theorem 3

$$\vdash \mathsf{set}\ (\mathsf{MAP}\ (\mathsf{abstr\_edge}\ AP)\ (\mathsf{concr\_trans}\ \phi))\ =\ \mathsf{trans}\ (\mathscr{P}(AP))\ \phi$$

Here trans is δ, computed for a specific alphabet, and abstr_edge applies transform_label to the lists of positive and negative atomic propositions of a concrete edge, and transforms the list of successors into a set.

Additionally we specify functions for adding nodes and edges to the graph representing the alternating automaton, add_state and add_edge. The function add_state is a wrapper around the generic function of the graph type for adding nodes that additionally decides whether or not a state should be final by checking if the formula is an "until"-formula. The function add_edge decides whether to add the edge to the true_labels field of the node, which it does if the set of successors is empty, or by using real edges in the graph. Because add_edge may be called for a node that is not in the graph, its return value uses the option type.

Using these auxiliaries, we define a recursive function called expand_graph (see Figure 5). It maintains a list of nodes to process and the current state of the graph. In every iteration the first element of the list is processed by computing its outgoing transitions with concr_trans and adding the successors and the edges to the graph. The list of nodes that still need to be processed is extended by the new successors if they have not been processed already. For a given formula φ, expand_graph is initially called with the list of formulae in $\overline{\varphi}$ (the set of initial states by Definition 6), and, as its first parameter, the graph containing only these formulae and no edges.

To show termination of expand_graph we use the fact that in the list of nodes to be processed we always remove one element f and replace it with its sucessors, all of which are subformulae of f. As the "subformulae of" relation is a partial order, this lets us use the multiset ordering to define a wellfounded order on the second argument of expand_graph that decreases in every iteration.

Concrete VWAA to GBA The second part of the concrete translation, from VWAA to GBA, takes a concrete alternating automaton as input and computes a

```
expand_graph g [] = SOME g
expand_graph g1 (f :: fs) =
  (let
     trans = concr_trans f ;
     sucs = nub (FOLDR (λ e pr. e.sucs ++ pr) [] trans) ;
     g2 = FOLDR (λ p g. add_state g p) g1 sucs ;
     g3 =
       FOLDR (λ e g?. monad_bind g? (add_edge f e))
         (SOME g2) trans ;
     new_to_process =
       FILTER
         (λ s. ¬MEM s (graph_states g1) ∧ s ≠ f ∧ ¬MEM s fs)
         sucs
   in
     case g3 of
     | NONE ⇒ NONE
     | SOME g ⇒ expand_graph g (new_to_process ++ fs))
```

Fig. 5: Concrete function implementing LTL to VWAA. The first argument is the graph of an alternating automaton and the second argument is the list of nodes that still need to be processed.

concrete GBA. The states of the GBA are labeled by lists of states of the VWAA. As the set of outgoing transitions of a GBA state depends on the transitions of the VWAA states in its label, we need to compute these from the input VWAA. We do this by defining a concrete version of the function abstr_transAA called get_concr_transAA.

To compute the transition of a GBA state labeled by a list of VWAA states L, we compute get_concr_transAA for every q in L and then apply a fold with our concrete version of $\otimes$ to the list of transitions. For every edge we then need to check for which of the final states f of the VWAA the conditions of T_f, given in Definition 7, apply. Remember that this includes a check whether there is a transition in $\delta(f)$ that does not contain f. To perform these checks more efficiently, we precompute the transitions for all final states of the VWAA.

Finally we need to remove all transitions that are not $\preceq$-minimal. To do this we define a concrete counterpart of $\preceq$. Having defined this relation, we find the minimal elements by comparing all the computed transitions of a state pairwise. We then add the successor states and the edges to the graph and extend the list of nodes to be processed by the new nodes.

Showing termination of this function is more involved than for the first part. The reason is that there is no partial order on the states of the GBA in general, indeed it can have non trivial cycles. To show termination we use the following insight: either the statespace of the graph grows, or it stays the same and the list of nodes to be processed becomes shorter. The first part is a wellfounded relation, as there is an upper bound on the total number of possible states, namely the powerset of the states of the alternating automaton, $\mathscr{P}(Q)$. Here we need to show that all new states computed by concr_trans are really in $\mathscr{P}(Q)$. If the statespace of the graph does not grow in some iteration of expand_graph, we know that all successors of the currently processed node must already have been processed. Thus the list of nodes to be processed gets shorter by one, as the current node is removed. Combining these two orders lexicographically leads to a wellfounded relation. The same approach to prove termination of a graph expansion algorithm was adopted in Schimpf *et al.* [15].

4.5 Verifying the concrete functions

After having defined our concrete automata types and concrete functions that implement the translations we show two things. First, they never return NONE on any reasonable input. For the VWAA to GBA translation we require a concrete alternating automaton as produced by the concrete LTL to VWAA translation. Second, applying the abstraction functions gives us exactly the abstract automata that we get by chaining the abstract translation function with the restriction to reachable states. For the LTL to VWAA translation we prove the following theorem, which essentially corresponds to the left hand side of Figure 2. The function concr_ltl2vwaa computes the list of initial states and calls expand_graph.

$$\begin{array}{l}\vdash \forall \varphi.\\ \quad \exists c_{AA}.\\ \qquad \mathsf{concr\_ltl2vwaa}\ \varphi\ =\ \mathsf{SOME}\ c_{AA}\ \wedge\\ \qquad \mathsf{abstr\_AA}\ c_{AA}\ =\ \mathsf{restr\_states}\ (\mathsf{ltl2vwaa}\ \varphi)\end{array}$$

To show the first part we need to show that we do not call add_edge for a node that is not in the graph, since this is the only possibility for expand_graph to return NONE (see Figure 5). We do this by showing that all nodes in the list that still have to be processed must have been added to the graph already.

The second part amounts to showing that, after applying the abstraction functions, the states, the transition function, the initial and the final states are equal to the corresponding fields in the result of the abstract translation.

Using Theorem 3 we show that for every state q that has already been processed it holds that all states that are one step reachable from q are either already in the graph, or in the list of nodes to be processed. Reachability here means the reflexive and transitive closure of δ. From this lemma follows that we will eventually include all reachable states of the abstract automaton. To show that only such states are included we again use Theorem 3 and show the invariant that every state in the graph is indeed reachable. In these two steps we use the assumption that the initial states are computed correctly, which we prove independently.

For the transition function we need to show that add_edge adds the edges computed by concr_trans in the intended way. To show this we show that for all nodes in the graph g that have been processed already, abstr_transAA g q is equal to $\delta(q)$.

The proofs for the abstractions of final and initial states amount to showing that the concrete computation of $\overline{\varphi}$ corresponds to the abstract function, and that exactly the nodes labeled by an "until" formula have $\mathsf{is\_final}$ set to *true*.

For the second part of the translation, from concrete VWAA to concrete GBA, we prove the following theorem, which corresponds to the right hand side of Figure 2:

$$\begin{aligned}
&\vdash \mathsf{concr\_ltl2vwaa}\ \varphi\ =\ \mathsf{SOME}\ c_{AA}\ \wedge\ a_{AA}\ =\ \mathsf{abstr\_AA}\ c_{AA}\ \Rightarrow\\
&\qquad \exists\, c_{GB}.\\
&\qquad\quad \mathsf{concr\_vwaa2gba}\ c_{AA}\ =\ \mathsf{SOME}\ c_{GB}\ \wedge\\
&\qquad\quad \mathsf{abstr\_GBA}\ c_{GB}\ =\ \mathsf{restr\_states}\ (\mathsf{vwaa2gba}\ a_{AA})
\end{aligned}$$

We have similar proof obligations here as in the previous case, we need to show that states, transition function and initial states are correctly abstracted. For the acceptance condition we show that for all f in $\mathsf{all\_acc\_frml}$ of the concrete automaton: if we collect all transitions in the concrete graph labeled by f, we get exactly the set T_f. Additionally, for every f in $\mathsf{all\_acc\_frml}$ we show that $T_f \in \mathcal{T}$, and for the other direction if $T_f \in T$, then f is in $\mathsf{all\_acc\_frml}$.

The states are handled by showing that the concrete computation of the transition function corresponds to the abstract definition and then using the same ideas as in the first translation step. For the transition function we have the advantage that it is more directly encoded in the edges of the graph. On the other hand we need to compute the transition functions of the VWAA states, that the GBA state is labeled by, correctly, and handle the minimization by $\preceq$. For the minimization we need to show that two concrete transitions are related by our concrete version of $\preceq$ if and only if their abstract counterparts are related by $\preceq$. This implies that we are removing the right transitions in the concrete function.

Translation to CakeML The CakeML ecosystem includes a general mechanism for translating (a subset of) HOL functions into provably equivalent CakeML ASTs (Myreen and Owens [13]). We use this technology to transform our concrete algorithm into CakeML syntax, to which we can then apply the CakeML compiler, generating assembly code. Under minimal assumptions (including: CakeML's model of the hardware corresponds to that of the chip that actually executes the code, and the correctness of the assembler and linker used to generate the final executable), the correctness of the CakeML compiler lets us conclude that this machine code will implement the algorithm exactly as written in the HOL formulation. In turn, the abstraction proofs described earlier then give us a high-assurance connection between the machine code that executes and the mathematical results of G&O.

At this stage, we embody our algorithms in a simple tool that parses an LTL formula on standard input, and prints out the two translated automata as (typically rather large) S-expressions. We have not benchmarked our executable's performance to any degree. Certainly, we are confident that CakeML-compiled code and a naïve representation of graphs/automata will not perform as well as hand-tuned C tools that have had extensive development. On the other hand, the development in HOL4 and CakeML gives us extremely high assurance that our tool is correct.

5 Related Work

The most complete verification effort of algorithms in the context of LTL model checking was done by Esparza *et al.* [6]. They describe a fully verified implementation of an LTL model checker in the Isabelle theorem prover. The work builds on a previously described verification [15] of the LTL to generalized Büchi automata translation which was introduced by Gerth *et al.* [8]. The algorithm uses a tableau construction and is more amenable to a direct verification as it does not include the intermediate step of alternating automata. The work has been extended to use Promela as input language to describe systems [14] and to use partial order reductions [1]. Additional optimization techniques for Büchi automata have been verified as independent functions in Schimpf and Smaus [16]. Another mechanization of a translation algorithm from LTL to automata was reported on in Esparza *et al.* [5]. The authors introduce a new algorithm targeting deterministic automata and emphasize the importance of interactive theorem provers, which allowed them to uncover errors in their original proofs.

One approach that has been developed to refine abstract definitions into efficient code is the Isabelle Refinement Framework [9, 10]. Both powerful and generic, it allows the refinement of abstract types into more efficient data structures. We believe that our rather custom abstraction would have been hard to achieve in this framework, as the structure of the abstract automata are quite different to the concrete ones, and multiple abstract details are encoded in the same concrete types. For example, consider the accepting edges of the GBA. While the abstract automaton provides all these edges in a set of sets, in the concrete world they are embedded in the graph using the edge labels.

Alternating automata in the context of interactive theorem proving were previously addressed by Merz [12]. This work mechanizes a proof of the closure of weak alternating automata under complementation, using winning strategies of logical games. As an application, Merz presents a translation from LTL into very weak alternating automata. The translation mechanized by Merz generates more states than G&O (all sub-formulae and negations *vs.* only temporal subformulae), and he does not address the second, exponential, translation to GBAs. This work also remains completely abstract, without mentioning concrete algorithms.

6 Conclusion

In this paper we have presented a formalization of the algorithm for translating LTL formulae into generalized Büchi automata presented in G&O, which uses very weak alternating automata as an intermediate representation.

We introduce an encoding of both alternating automata and generalized Büchi automata in a compilable, generic graph type that uses an efficient lookup structure. This is especially interesting for alternating automata, as they are a powerful computational model leading to elegant algorithms, *e.g.*, Vardi [18].

To cope with the complexity of the algorithm, we divide the formalization into an abstract and a concrete part. In the abstract part we mechanize the proofs and show correctness of the translation as it is presented in G&O. The correspondence between the abstract and concrete models is established using abstraction functions that map concrete automata to abstract ones. We implement the algorithm on our concrete types and show that applying the abstractions to the resulting automata leads to the automata given by the abstract translation.

This approach turned out to be fruitful: we were able to reproduce the abstract correctness results fairly quickly. Not having to additionally cope with arguments about concrete data structures, termination arguments and details concerning our graph type, made a big difference. We would like to extend our ideas to include optimization steps in the translation, by showing independent correctness in the abstract world and efficiently embedding them in the expansion algorithm. So far, efficiency has not been a big concern for us; rather we have focused on producing verified code for the algorithm in G&O. In future work we would like to optimize the code and provide an empirical comparison to existing tools.

Finally we use the CakeML compiler to produce fully verified code implementing our concrete functions. This step significantly strengthens the confidence we can have in the machine code, as we do not have to trust a standard compiler. Translation of LTL formulae into automata is only one part of a complete model checker, but our experience suggests that an extremely high assurance model checker embodying sophisticated optimizations is entirely feasible.

References

1. Julian Brunner and Peter Lammich. Formal verification of an executable LTL model checker with partial order reduction. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, pages 307–321, Cham, 2016. Springer International Publishing.
2. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, Oct 1992.
3. Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In Patrice Godefroid, editor, *Model Checking Software*, pages 169–184, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
4. Martin Erwig. Functional programming with graphs. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 52–65. ACM, 1997.
5. Javier Esparza, Jan Kretínský, and Salomon Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016.
6. Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 463–478, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
7. Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
8. Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.
9. Peter Lammich. Automatic Data Refinement. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 84–99, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
10. Peter Lammich and Thomas Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 166–182, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
11. Christof Loding and Wolfgang Thomas. Alternating automata and logics over infinite words. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, pages 521–535, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
12. Stephan Merz. Weak alternating automata in Isabelle/HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000 Portland, OR, USA, August 14–18, 2000 Proceedings*, pages 424–441, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

13. Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012.
14. René Neumann. Using Promela in a fully verified executable LTL model checker. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, pages 105–114, Cham, 2014. Springer International Publishing.
15. Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 424–439, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
16. Alexander Schimpf and Jan-Georg Smaus. Büchi automata optimisations formalised in Isabelle/HOL. In Mohua Banerjee and Shankara Narayanan Krishna, editors, *Logic and Its Applications: 6th Indian Conference, ICLA 2015, Mumbai, India, January 8-10, 2015. Proceedings*, pages 158–169, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
17. Moshe Y. Vardi. Nontraditional applications of automata theory. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 575–597, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
18. Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In William McCune, editor, *Automated Deduction—CADE-14*, pages 191–206, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.