



Efficient translation of LTL formulae into Büchi automata

**Dimitra Giannakopoulou
Flavio Lerda**

RIACS Technical Report 01.29

June 2001

Efficient translation of LTL formulae into Büchi automata

Dimitra Giannakopoulou, RIACS

Flavio Lerda, CMU

RIACS Technical Report 01.29

June 2001

Model checking is a fully automated technique for checking that a system satisfies a set of required properties. With explicit-state model checkers, properties are typically defined in linear-time temporal logic (LTL), and are translated into Büchi automata in order to be checked. This report presents how we have combined and improved existing techniques to obtain an efficient LTL to Büchi automata translator. In particular, we optimize the core of existing tableau-based approaches to generate significantly smaller automata. Our approach has been implemented and is being released as part of the Java PathFinder software (JPF), an explicit state model checker under development at the NASA Ames Research Center.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA).

This report is available online at <http://www.riacs.edu/trs/>

Efficient translation of LTL formulae into Büchi automata

Dimitra Giannakopoulou

RIACS/USRA
NASA Ames Research Center
Moffett Field, CA 94035-1000
Email: dimitra@email.arc.nasa.gov

Flavio Lerda

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
Email: lerda@cmu.edu

Abstract

Model checking is a fully automated technique for checking that a system satisfies a set of required properties. With explicit-state model checkers, properties are typically defined in linear-time temporal logic (LTL), and are translated into Büchi automata in order to be checked. This report presents how we have combined and improved existing techniques to obtain an efficient LTL to Büchi automata translator. In particular, we optimize the core of existing tableau-based approaches to generate significantly smaller automata. Our approach has been implemented and is being released as part of the Java PathFinder software (JPF), an explicit state model checker under development at the NASA Ames Research Center.

1 Introduction

The use of LTL-based specifications in model checking is widespread. Many tools, including Java PathFinder, developed at the NASA Ames Research Center [1], and SPIN, from Bell Labs [2], after translating LTL formulae into Büchi automata, perform the verification using algorithms based on the one presented in [3]. These algorithms are linear in the size of the Büchi automata; however the Büchi automaton corresponding to an LTL formula may, in the worst case, be exponential in the size of the formula, making the model checking effort exponential in the size of the original formula. Since finding the optimal sized Büchi automaton is a PSPACE-hard problem [4], the translation process becomes crucial in determining the size of the automata used for verification. Moreover, it is important for automata used for verification to be as small as possible, since memory is a major concern in model checking.

We will present a tool, LTL2BUCHI, that we developed at the NASA Ames Research Center to translate LTL formulae into Büchi automata. LTL2BUCHI has been written in Java for readability, portability, but also to be integrated into Java PathFinder. The work presented here is however completely general, since the translation does not need to take into account the syntax of the atomic predicates, which is the only part specific to Java PathFinder. LTL2BUCHI is an efficient translator, which generates very concise automata for most LTL properties of practical interest. It achieves this by combining but also *improving* existing techniques.

This report is a detailed documentation of the algorithms used in the various stages of the translation process, with an emphasis on the novel aspects of our approach. It also contains some early results from the comparison of our approach to existing ones. Specifically, the paper is organized as follows. First, we provide some background information in Section 2, followed by a description the algorithm used in our tool in Section 3, and an informal correctness argument in Section 4. Section 5 presents experimental results and comparisons with other LTL to Büchi translators, and discusses directions for future work.

2 Background

2.1 Linear Temporal Logic (LTL)

In this work, LTL is used to express temporal properties of a system for model checking. Given a set of atomic propositions \wp , a well-formed LTL formula is constructed using Boolean connectives (\neg , \wedge , \vee , \rightarrow) and the temporal operators **X** (next), **U** (strong until), **V** (release – dual of **U**), **F** (eventually), **G** (always), **W** (weak until), **M** (strong release – dual of **W**).

Note that, although we present the operator **X** for completeness in this section, in what follows we use only the next-free variant of LTL, namely LTL-X. This is typical in model checking, because LTL-X is guaranteed to be insensitive to stuttering [5]. This property is important because it avoids the notion of an absolute next state. The next time operator (**X**) is misleading, because users naturally tend to assume some level of abstraction on the state of a program. In the rest of this paper, the next-free variant of LTL is implied whenever we refer to LTL, unless explicitly noted.

Definition 1 (grammar). *The set $L_{LTL, \wp}$ of the well-formed LTL formulae over a finite set of atomic propositions \wp is the language defined by the following grammar:*

$$G = \langle T, N, P, \varphi \rangle$$

where

$$T = \wp \cup \{ \neg, \wedge, \vee, \rightarrow, \mathbf{X}, \mathbf{U}, \mathbf{V}, \mathbf{F}, \mathbf{G}, \mathbf{W}, \mathbf{M}, \text{true}, \text{false} \}$$

$$N = \{ \varphi \}$$

$$P = \{ \varphi \rightarrow p \mid \forall p \in \wp \} \cup$$

$$\begin{aligned} & \{ \varphi \rightarrow \text{true}, \\ & \varphi \rightarrow \text{false}, \\ & \varphi \rightarrow \neg \varphi, \\ & \varphi \rightarrow \varphi \wedge \varphi, \\ & \varphi \rightarrow \varphi \vee \varphi, \\ & \varphi \rightarrow \varphi \rightarrow \varphi, \\ & \varphi \rightarrow \mathbf{X} \varphi, \\ & \varphi \rightarrow \varphi \mathbf{U} \varphi, \\ & \varphi \rightarrow \varphi \mathbf{V} \varphi, \\ & \varphi \rightarrow \mathbf{F} \varphi, \\ & \varphi \rightarrow \mathbf{G} \varphi, \\ & \varphi \rightarrow \varphi \mathbf{W} \varphi, \\ & \varphi \rightarrow \varphi \mathbf{M} \varphi \} \end{aligned}$$

Some of the operators in the above grammar are derived from others as defined below:

Definition 2 (derived operators). *For $\varphi, \psi \in L_{LTL, \wp}$, derived operators are defined as follows:*

$$\begin{aligned} \varphi \wedge \psi &\equiv \neg (\neg \varphi \vee \neg \psi) \\ \varphi \rightarrow \psi &\equiv (\neg \varphi \vee \psi) \\ \varphi \mathbf{V} \psi &\equiv \neg (\neg \varphi \mathbf{U} \neg \psi) \\ \mathbf{F} \varphi &\equiv \text{true} \mathbf{U} \varphi \\ \mathbf{G} \varphi &\equiv \neg \mathbf{F} \neg \varphi = \text{false} \mathbf{V} \varphi \\ \varphi \mathbf{W} \psi &\equiv (\varphi \mathbf{U} \psi) \vee \mathbf{G} \varphi \\ \varphi \mathbf{M} \psi &\equiv \neg (\neg \varphi \mathbf{W} \neg \psi) \end{aligned}$$

An interpretation of an LTL formula is an infinite word w over the power set of the set of propositions \wp . An LTL formula φ defines a language $L_{\varphi, \wp}$ which is the set of all the possible interpretations of φ . We define an interpretation of an LTL formula using only non-derived operators.

Definition 3 (infinite word). An infinite word $w \in \Sigma^\omega$ over an alphabet Σ , is an infinite sequence of symbols in Σ , i.e., $w = x_0x_1x_2\dots$ such that $\forall i \in \mathbb{N}: x_i \in \Sigma$. We write w_i for the suffix of w starting at i .

Definition 4 (interpretation). An infinite word $w = x_0x_1x_2\dots$ over the alphabet $\Sigma = 2^\wp$ is an interpretation of an LTL formula φ , written $w \models \varphi$, if the following hold:

$$\begin{aligned} w \models p &\Leftrightarrow p \in x_0 \\ w \models \neg \varphi &\Leftrightarrow \neg w \models \varphi \\ w \models \varphi \vee \psi &\Leftrightarrow (w \models \varphi) \vee (w \models \psi) \\ w \models \varphi \mathbf{U} \psi &\Leftrightarrow \exists i \in \mathbb{N}, w_i \models \psi \text{ and } \forall 0 \leq j < i, w_j \models \varphi \\ w \models \mathbf{X} \varphi &\Leftrightarrow w_1 \models \varphi \end{aligned}$$

2.2 Büchi Automata (BA)

Definition 5 (Büchi automaton). A Büchi automaton B is a 5-tuple:

$$B = \langle S, A, \Delta, q_0, F \rangle$$

where:

- S is a finite set of states
- A is a finite set of labels
- $\Delta \subseteq S \times A \times S$ is a transition relation
- $q_0 \in S$ is the initial state
- $F \subseteq S$ is a set of accepting states.

Definition 6 (execution). An execution of the Büchi automaton B on an infinite word $w = a_1a_2a_3\dots$ over alphabet A is an infinite word $\sigma = q_0q_1q_2\dots$ over alphabet S , such that: $\forall i \in \mathbb{N}^+, q_i \in \Delta(q_{i-1}, a_i)$.

Definition 7 (accepted words). An infinite word w over alphabet A is accepted by a Büchi automaton B , if and only if there exists an execution of B on w that contains at least one element of F an infinite number of times.

2.3 Generalized Büchi Automata (GBA)

Definition 8 (generalized Büchi automaton). A generalized Büchi automaton GB is a 5-tuple:

$$GB = \langle S, A, \Delta, q_0, F \rangle$$

where:

- S is a finite set of states
- A is a finite set of labels
- $\Delta \subseteq S \times A \times S$ is a transition relation
- $q_0 \in S$ is an initial state
- $F \subseteq 2^S$ is a set of sets of accepting states.

Definition 9 (execution). An execution of the generalized Büchi automaton GB on an infinite word $w = a_1a_2a_3\dots$ over alphabet A is an infinite word $\sigma = q_0q_1q_2\dots$ over alphabet S , such that: $\forall i \in \mathbb{N}^+, q_i \in \Delta(q_{i-1}, a_i)$.

Definition 10 (accepted words). An infinite word w over the alphabet A is accepted by a generalized Büchi automaton GB, if and only if there exists an execution of GB on w that contains at least one element from each element of the sets in F an infinite number of times.

2.4 Transition-based Generalized Büchi Automata (TGBA)

Definition 11 (transition-based generalized Büchi automaton). A transition-based generalized Büchi automaton TGB is a 5-tuple:

$$\text{TGB} = \langle S, A, T, q_0, F \rangle$$

where:

S is a finite set of states

A is a finite set of labels

$T \subseteq S \times A \times S$ is a set of transitions

$q_0 \in S$ is an initial state

$F \subseteq 2^T$ is a set of sets of accepting transitions.

Definition 12 (execution). An execution of transition-based generalized Büchi automaton TGB on an infinite word $w = a_1 a_2 a_3 \dots$ over alphabet A is an infinite word $\sigma = (q_0 a_1 q_1)(q_1 a_2 q_2) \dots$ over alphabet $S \times A \times S$, such that: $\forall i \in \mathbb{N}^+, (q_{i-1}, a_i, q_i) \in T$.

Definition 13 (accepted words). An infinite word w over the alphabet A is accepted by a transition-based generalized Büchi automaton TGB, if and only if there exists an execution of TGB on w that contains at least one element from each element of the sets in F an infinite number of times.

3 Algorithm

Similarly to others [4, 6], our translation process consists of three stages (see Figure 1).

- Formula rewriting.
- Translation of LTL formula into a generalized Büchi automaton. Specifically, our algorithm generates a transition-based generalized Büchi automaton (TGBA). We refer to this as the “core” of the translation process.
- Conversion of the generalized Büchi automaton into a Büchi automaton (we refer to this process as *degeneralization*).

Similarly to [7], we perform various optimizations, described in detail later, at each step of the translation to keep the automata generated small in size.

One of the major improvements of our approach as compared to other tableau-based translations ([8, 9]) comes from the “core” of the translation. Our algorithm is based on that by [9]. However, due to the fact that it generates transition-based generalized Büchi automata, rather than classical state-based ones, it merges more states than other approaches can. For example, as will be described in Section 3.2.2, our algorithm may merge states, which, with state-based approaches, would be characterized by different/contradicting literals, or would belong to different accepting sets, and therefore considered not equivalent.

For model checking, the final result has to be a simple (non-generalized) Büchi automaton with accepting states rather than transitions. In the third stage, our approach transforms the automaton

obtained from LTL2TGBA from transition-based to state-based and from generalized to non-generalized in one single step. This allows us to limit the growth of the number of states from the generalized to the non-generalized Büchi automaton.



Figure 1. Stages of the translation process

3.1 Formula Rewriting

This stage applies rewriting rules to the original formula to obtain an equivalent formula that may result in a smaller automaton. We have implemented a simple rewriting engine that allows us to configure the rules using a text file, and have used several rewriting rules presented in [4, 6]. The rules we are currently using are the following:

$p \wedge p = p$ $p \wedge \text{true} = p$ $p \wedge \text{false} = \text{false}$ $p \wedge !p = \text{false}$ $p \vee p = p$ $p \vee \text{true} = \text{true}$ $p \vee \text{false} = p$ $p \vee !p = \text{true}$ $(\mathbf{X} p) \mathbf{U} (\mathbf{X} q) = \mathbf{X} (p \mathbf{U} q)$ $(p \mathbf{V} q) \wedge (p \mathbf{V} r) = p \mathbf{V} (q \wedge r)$ $(p \mathbf{V} r) \vee (q \mathbf{V} r) = (p \vee q) \mathbf{V} r$ $(\mathbf{X} p) \wedge (\mathbf{X} q) = \mathbf{X} (p \wedge q)$	$\mathbf{X} \text{true} = \text{true}$ $p \mathbf{U} \text{false} = \text{false}$ $\mathbf{G} \mathbf{F} p \vee \mathbf{G} \mathbf{F} q = \mathbf{G} \mathbf{F} (p \vee q)$ $\mathbf{F} \mathbf{X} p = \mathbf{X} \mathbf{F} p$ $\mathbf{G} \mathbf{G} \mathbf{F} p = \mathbf{G} \mathbf{F} p$ $\mathbf{F} \mathbf{G} \mathbf{F} p = \mathbf{G} \mathbf{F} p$ $\mathbf{X} \mathbf{G} \mathbf{F} p = \mathbf{G} \mathbf{F} p$ $\mathbf{F} (p \wedge \mathbf{G} \mathbf{F} q) = (\mathbf{G} p) \wedge (\mathbf{G} \mathbf{F} q)$ $\mathbf{G} (p \vee \mathbf{G} \mathbf{F} q) = (\mathbf{G} p) \vee (\mathbf{G} \mathbf{F} q)$ $\mathbf{X} (p \wedge \mathbf{G} \mathbf{F} q) = (\mathbf{X} p) \wedge (\mathbf{G} \mathbf{F} q)$ $\mathbf{X} (p \vee \mathbf{G} \mathbf{F} q) = (\mathbf{X} p) \vee (\mathbf{G} \mathbf{F} q)$
--	--

Note that, rewriting rules for the \mathbf{X} operator will not be used in practice since we expect that only next-time free LTL formulae will be of interest to developers – the rules have just been added to the file for completeness. We have used rewriting rules not only for temporal operators but for standard Boolean logic as well in order to simplify the formula as soon as possible.

Since the next step of the translation needs the formula to be expressed in negated normal form and to contain only the operators \neg , \wedge , \vee , \mathbf{X} , \mathbf{U} , and \mathbf{V} , we represent (internally) the LTL formula and both sides of the rewriting rules in negation normal form using only the above set of operators. This is performed by substituting all other operators based on the definitions provided in Section 2.1. For instance, in what follows, rewriting rule (i) is internally represented as rewriting rule (ii):

- (i) $\mathbf{G} \mathbf{F} p \vee \mathbf{G} \mathbf{F} q = \mathbf{G} \mathbf{F} (p \vee q)$
(ii) $((\text{true} \mathbf{U} p) \mathbf{U} \text{false}) \vee ((\text{true} \mathbf{U} q) \mathbf{U} \text{false}) = (\text{true} \mathbf{U} (p \vee q)) \mathbf{U} \text{false}$

Even though this seems as a more complicated way to store the rule, it has some advantages. For example, in form (i), the above rule can be applied directly to the following LTL formula:

$$\mathbf{GF} (a \mathbf{U} b) \vee \mathbf{GF} c \quad \text{to obtain:} \quad \mathbf{GF} ((a \mathbf{U} b \vee c))$$

Form (ii) can also be applied easily to the internal representation of the original formula, which is:

$$((\text{true } \mathbf{U} (a \mathbf{U} b)) \mathbf{U} \text{false}) \vee ((\text{true } \mathbf{U} c) \mathbf{U} \text{false})$$

and which, based on simple pattern matching with form (ii) is rewritten as:

$$(\text{true } \mathbf{U} (a \mathbf{U} b \vee c)) \mathbf{U} \text{false}$$

On the other hand, in its first form, the rewriting rule above cannot be applied directly (without using the definition of the derived operators) to formula: “ $\mathbf{G} (\text{true } \mathbf{U} (a \mathbf{U} b)) \vee \mathbf{GF} c$ ”, which is clearly equivalent to the previous one. By using the internal representations of both the formula and the rewriting rule, there is no need for any special action in order to perform the rewriting. Note that our tool allows expressing the rewriting rules using the complete LTL grammar to make encoding easier, but translates them appropriately at loading time.

3.2 LTL2TGBA

Our algorithm (and its presentation) is based on that by [8], and includes the improvements proposed by [9]. As discussed, however, it introduces several modifications. In this section, we first present a brief overview of these modifications for readers that are familiar with tableau-based approaches for the “core” translation. Following that, the algorithm is presented in detail.

The translation is performed on a *Node*, which is a similar data structure to that of [8]. The information contained in a node is used to generate states and transitions of the final automaton. It differs from the structure of [8] in that field *Old* contains only the literals that have been processed. From the non-literals in *Old*, only \mathbf{U} formulae and formulae that correspond to the right-hand side formulae of a \mathbf{U} formula need to be stored, in order to compute the accepting sets of states of the automaton. We store such information in bitmaps, as follows.

The set of accepting sets to which a Node belongs is represented as a bitmap B of size $\#\{\text{accepting sets}\}$. If location i of B is set, then the Node belongs to accepting set i .

As mentioned, operators \mathbf{G} , \mathbf{F} , \mathbf{W} and \mathbf{M} are simply abbreviations, so our algorithm transforms them appropriately during parsing of the formulae. As defined by [8], there are as many accepting sets in the automaton generated, as there are \mathbf{U} sub-formulae in the formula ϕ that is being translated. We assign a unique index (starting from 0) to each \mathbf{U} sub-formula, which reflects the location of the corresponding accepting set in the bitmap described above. The original formula is represented using a directed a-cyclic graph (dag), where common subformulae are stored only once. If a common subformula contains operator \mathbf{U} , it will also be counted only once when determining the number of accepting sets.

In our approach, each Node has two additional fields, *Untils* and *Right_of_Untils*, which record, for each \mathbf{U} formula, whether it has been processed in the current Node, or whether its right-hand sub-formula has been processed in the current Node, respectively. The accepting sets of a Node are then obtained by performing bit-wise operations on these two bitmaps, as described in detail below. This is more efficient than computing the accepting information based on syntactic implications as performed in [9], since the necessary information is obtained anyway during the translation process.

In the automata that we generate, states are not labeled. Transitions carry labels both of the literals that need to be checked for those transitions to fire, and of the accepting sets to which these transitions belong. As a result, several transitions may connect two states. Transitions are merged if they agree both on the literals that label them, and on the accepting sets to which they belong. Information about states and transitions of such automata is contained in the Node structure. In essence, a Node represents a State of the automaton. Its *Old* information is used to label all the

transitions that lead to that state, and its *Untils* and *Right_of_Untils* bitmaps to define the accepting sets of these transitions.

Note that, as with all other similar algorithms, ours also handles formulae in negation normal form, i.e. all the negations are pushed inside until they only precede atomic propositions. In the following, we describe the algorithm and the data structures it manipulates in detail.

3.2.1 Data Structures

The basic data structure that the automaton construction algorithm manipulates is the *Node*, which contains the following fields:

NodeId: A unique node id. Needed after the end of the construction algorithm. It uniquely identifies the state represented by the node. It gets assigned from a pool class that starts assigning from 1. Id 0 is reserved for the *initial* state.

Incoming: A set of node/state Ids of which this node is an immediate successor in the resulting automaton.

ToBeDone: A set of formulae that must hold at the current state and have not yet been processed.

Untils: Bitmap of size equal to the number of **U** sub-formulae of the formula that is being translated. A bit is set if the corresponding sub-formula has been processed in the node.

Right_of_untils: A bitmap that records, for each **U** sub-formula, whether its right component has been processed in the node.

Old: Set of literals that must hold at the current node.

Next: Formulae that must hold in all states that are immediate successors of states satisfying the properties in *Old*.

The data structure *State* represents a state of the Büchi automaton generated. It has the following fields:

StateId: Similar to NodeId. As states are generated from Nodes, a state gets its Id from the corresponding NodeId.

Transitions: A set containing all transitions that lead to the this state.

Next: Formulae that must hold in all states that are immediate successors of this state.

Each *Transition* is a data structure that contains the following fields:

Source: Set of Ids of source states of the transition. Notice that our algorithm often groups transitions with the same labels that lead to the same state. As a result, the Source is a *set* of state Ids.

Label: The set of literals that must hold for the transition to be triggered.

Accepting: A bitmap that records to which accepting sets the transition belongs.

We keep a list, *states*, of the computed states of the Büchi automaton that is being generated. We denote the field *ToBeDone* of the node *q* by *q.ToBeDone*, and similarly for other fields. Moreover, we denote the equivalence class with node *q* at its head (at the head of the list) as *equiv(q)*.

```

1  /** function expand is a method of class Node***/
2  States_Set expand (States_Set states) {
3
4  if ToBeDone is empty { // node has been fully processed
5      if  $\exists$  ST  $\in$  states s.t. ST.Next equals This.Next { // this node equivalent to ST
6          ST.merge(This);
7          return states;
8      } // end if – else below refers to “there is no equivalent node already in Nodes_Set”
9      else { // processed node to be added to states
10         NS = new State(This);
11         states = states  $\cup$  {NS};
12         create NewNode with new NodeId, Old and Next empty, bitmaps unset
13         and (Incoming = This.NodeId, ToBeDone = This.Next);
14         return NewNode.expand(states);
15     }
16 } else { // ToBeDone is not empty, so keep processing
17     let next_formula  $\in$  ToBeDone;
18     remove next_formula from ToBeDone;
19
20     if (next_formula is the right hand child of a U formula)
21         This.Right_of_untils.or(next_formula.get_rightOfWhichUntils());
22
23     if (testForContradictions(next_formula)) // node contains contradiction
24         return states; // so gets discarded
25
26     if (isRedundant(next_formula)) // formula is redundant
27         return This.expand(states); // so no need to process it
28
29     if (next_formula is a U formula)
30         This.Untils.set(next_formula.get_untils_index());
31
32     // no contradictions, and formula is not redundant, so we process it
33     if (next_formula is not a literal) {
34         if (next_formula is a 'U', 'V' or 'v' formula) {
35             Node2 = This.Split(next_Formula); // node split in two
36             return Node2.expand(This.expand(states)); // expand depth-first
37         }
38         if (next_formula is a ' $\phi \wedge \psi$ ' formula) {
39             if ( $\phi \notin$  Old) ToBeDone = ToBeDone  $\cup$  { $\phi$ };
40             if ( $\psi \notin$  Old) ToBeDone = ToBeDone  $\cup$  { $\psi$ };
41             return This.expand(states);
42         }
43     } else { // next formula is a literal
44         add next_formula to Old; // just require formula true
45         return This.expand(states);
46     }
47 } // end of “else ToBeDone not empty”

```

Figure 2. Node expansion algorithm

3.2.2 Algorithm

An abstract syntax graph is originally generated for the formula φ that is being processed. The parse graph is similar to an abstract syntax tree, but is a dag where equal formulae appear only once, i.e., they are represented by a single node in the graph. That allows fast comparison of formulae in the subsequent phases of the algorithm, based on equality of their references.

Each node of the graph represents a sub-formula of φ (with the root referencing φ itself). For each (sub)-formula ψ , the following information is recorded. If ψ is a U-formula, we record the unique index (*untils_index*) of the accepting set that ψ defines. This index also reflects the index of the formula in the bitmap *Untils* of each Node object. The method *get_untils_index()* returns the index of the formula in the *Untils* bitmaps. Similarly, if ψ is the right-hand side of one or more U-formulae μ_i , we record for ψ in the bitmap *RightofWhichUntils* the indices of all μ_i formulae. In other words, if bit i is set in the *RightofWhichUntils* field of formula ψ , it means that ψ is the right-hand formula of the U-formula that defines the i^{th} accepting set, and corresponds to location i in the *Untils* bitmaps. The method *get_rightOfWhichUntils ()* returns the bitmap *RightofWhichUntils*.

The algorithm for transforming a formula φ starts by creating the initial Node *INIT* with *NodeId* = 0, with *Next* = $\{\varphi\}$, and with all other fields empty. The object *states* of type *States_Set* that will hold the states of the generated automaton is initially empty. The transformation of φ is then performed by expanding node *INIT*, i.e. by calling *INIT.expand(states)*.

Let us at this stage describe how the expansion method works. The line numbers in the following description refer to the algorithm that appears in **Error! Reference source not found.**

With the current node N , the algorithm first checks if there are unprocessed obligations left in *ToBeDone*. We will examine two cases: if 1) there are *none* 2) there are some obligations left in *ToBeDone*.

Case 1 – there are no obligations left in *ToBeDone*

This case is illustrated in lines 4-15. The fact that *ToBeDone* is empty shows that the current node is fully processed and ready to become a state (be added to *states*). If the state that this node represents is equivalent to an existing state in *states*, then the node will be merged with that state (line 6). In our context where states of the Büchi automaton generated are not labeled, two states are equivalent if they have the same obligations in their *Next* fields (line 5). A node is merged with a state as follows:

```
/** function merge is a method of class State***/
void merge (Node nd) {
    // satisfies accepting condition if corresponding U formula has not been processed,
    // or if the corresponding right hand child of the formula has been processed [8]
    acc = (bitwise_not (nd.Untils)) bitwise_or (nd.Right_of_untils);

    if ( $\exists$  TR  $\in$  This.Transitions s.t.
        (TR.Label equals nd.Old) and (TR.Accepting equals acc)) {
        TR.Source = TR.Source  $\cup$  {Incoming}
    } else {
        This.Transitions = This.Transitions  $\cup$  {new Transition with:
            Source = nd.Incoming, Label = nd.Old, Accepting = acc}
    } // end else
}
```

Notice from the above that the *Old* field of the Node is used for labeling all transitions that lead to the new state generated, and that the *Untils* and *Right_of_Untils* fields of the Node are used to compute the accepting sets to which the transition belongs.

The way accepting conditions are evaluated in the above procedure reflects the following fact [8]: for each **U** sub-formula φ of the formula being translated, a transition leading to a state *st* is accepting with respect to φ if either φ does not need to hold in *st* (i.e., φ has not been processed), or the right-hand side formula of φ has been processed. This result is obtained by computing the bitwise-or of the bitwise-negated *Untils* bitmap with the *Right_of_untils* bitmap.

If the state that this node represents is not equivalent to any existing state in *states*, then a new state is created (line 10) with the following constructor, and added to set *states* of the automaton:

```
State (Node nd) {
    StateId = NodeId;
    acc = (bitwise_not (nd.Untils)) bitwise_or (nd.Right_of_untils);

    Transitions = {new Transition with:
        Source = nd.Incoming, Label = nd.Old, Accepting = acc}

    Next = nd.Next;
}
```

Moreover, a new node *NewNode* is created as the immediate successor of the node that has just been processed. *NewNode* has the *NodeId* of the node that was processed in its *Incoming* field, and its obligations in *ToBeDone* are the obligations that the processed node holds in its *Next* field. In other words, after a node has been processed, its future obligations are delegated to its immediate successors. The *Untils* and *Right_of_untils* bitmaps of *NewNode* have all their bits cleared, and its *Old* and *Next* fields are empty.

Case 2 – there are obligations left in *ToBeDone*

This case is illustrated in lines 16-46. If the current node still contains obligations in *ToBeDone*, a formula *next_formula* is removed from this set. Since this formula is being processed, if it is the right-hand formula of some **U**-formulae, we update the corresponding fields in the *Right_of_Untils* bitmap. Then we check whether this formula contradicts any information already contained in this node (to improve readability, we defer the description of the algorithms for *testForContradictions* and *isRedundant* to the next section). If a contradiction occurs, it means that this node must be discarded. If no contradiction occurs, then we check whether the formula is redundant, in which case we simply do not need to process it [9]. Only then (i.e., if the formula is not redundant) do we check whether the formula is a **U**-formula, in which case we update the corresponding field in the *Untils* bitmap. As will be discussed in Section 4, the *Untils* bitmap does not need to be updated for redundant **U**-formulae.

If the formula is neither redundant, nor contradicts existing node information, then it gets processed as follows:

When *next_formula* is a literal, then the formula is simply added to field *Old* of the node (lines 42-45). When *next_formula* is not a literal, the current node is either split in 2 nodes (lines 33-36) or not split (lines 37-41), and new formulae may be added to the fields *ToBeDone* and *Next*. (Note that when we split a node, for efficiency reasons, we do not create two new nodes, but modify the current one, and create an additional node.) The exact actions performed depend on the form of *next_formula* and are the following:

- $next_formula = \varphi \wedge \psi$.

Then both ϕ and ψ are added to *ToBeDone* because they both need to be true for the formula to hold.

- *next_formula* is in either of the forms: $\phi \vee \psi$, $\phi \cup \psi$, $\phi \vee \psi$.

There are two alternative ways of making these formulae true. So the node is split into two nodes, each representing one way of making the formula true. For $\phi \vee \psi$, ϕ is added to *ToBeDone* of one node, and ψ to that of the other. For $\phi \cup \psi$, ϕ is added to *ToBeDone* and $\phi \cup \psi$ to *Next* of one node, and ψ is added to *ToBeDone* of the other. This splitting can be explained by observing that $\phi \cup \psi$ is equivalent to $\psi \vee (\phi \wedge X(\phi \cup \psi))$. For $\phi \vee \psi$, ψ is added to *ToBeDone* of both nodes, ϕ is added to *ToBeDone* of one node, and $\phi \vee \psi$ to *Next* of the other. This splitting can be explained by observing that $\phi \vee \psi$ is equivalent to $\psi \wedge (\phi \vee X(\phi \vee \psi))$.

The splitting algorithm is illustrated in Figure 3. Table 1 illustrates, for the types of formulae that cause a node to split, the formulae that are added to various fields of the resulting nodes (although \wedge formulae are not split, we include an entry in the table, because the fields in this table are also used for the definition of syntactic implication in Section 3.2). For example, the formulae in *New1* and *New2* are added to the *ToBeDone* field of the first and second resulting node, respectively. Moreover, *Next1* is added to the *Next* field of the first resulting node. Note that the first node of a split is just the initial node, modified accordingly.

The copies are processed in DFS order, i.e., when expansion of the current node and its successors are finished, the expansion of the second copy and its successors is started.

Note that a formula is only added to *ToBeDone* if it does not exist in *Old* – hence the fact that we take $(New(form) \setminus Old)$. This is purely for efficiency, that is, to avoid processing a formula that has already been processed. Also, the fields *Incoming*, *ToBeDone*, *Old*, and *Next* of each node are sets, and therefore contain no duplicates.

Table 1: Definition of New and Next functions for non-literals

FORM	New1(FORM)	Next1(FORM)	New2(FORM)
$\phi \cup \psi$	$\{\phi\}$	$\{\phi \cup \psi\}$	$\{\psi\}$
$\phi \vee \psi$	$\{\psi\}$	$\{\phi \vee \psi\}$	$\{\phi, \psi\}$
$\phi \vee \psi$	$\{\phi\}$	\emptyset	$\{\psi\}$
$\phi \wedge \psi$	\emptyset	\emptyset	$\{\phi, \psi\}$

Testing for contradictions and redundancies

The checks for contradiction and redundancy follow the approach of [9], which we slightly modify due to the fact that we keep the formulae that must hold at a node and those that must hold at its successors in different sets. They are both based on deriving the set of formulae $SJ(A, B)$ that are syntactically implied from sets of formulae A and B , where B represents formulae that have to hold at the next state. We use the following inductive definition:

1. $TRUE \in SJ(A, B)$,
2. $\mu \in SJ(A, B)$, if $\mu \in A$,
3. $\mu \in SJ(A, B)$, if μ is not a literal and either of the following hold:
 - $(New1(\mu) \subseteq SJ(A, B))$ and $(Next1(\mu) \subseteq B)$
 - $New2(\mu) \subseteq SJ(A, B)$.

Based on this definition, a formula φ contradicts a node nd , if $\neg\varphi \in SI(nd.Old, nd.Next)$. In other words, $nd.testForContradictions(\varphi)$ returns true if $\neg\varphi \in SI(nd.Old, nd.Next)$. A formula φ is redundant for a node nd , if $\varphi \in SI(nd.Old, nd.Next)$, and additionally, if φ is $\mu \mathbf{U} v$ (that is, φ is a **U**-formula), then $v \in SI(nd.Old, nd.Next)$. As mentioned in [9], the special attention to the right-hand arguments of **U**-formulae is for avoiding discarding information required to define accepting conditions. So to summarize, $nd.isRedundant(\varphi)$ returns true if $(\varphi \in SI(nd.Old, nd.Next))$ and (either φ is not a **U**-formula, or φ 's right hand argument $v \in SI(nd.Old, nd.Next)$).

*/** split is a method of class Node. It splits a node into two, using info in table */*

```

1 | Node split (Formula form) {
2 |     create Node2 with new Id s.t.
3 |         ( Node2.Incoming = Incoming,
4 |           Node2.ToBeDone = ToBeDone  $\cup$  (New2(form) \ Old),
5 |           Node2.Untils and Node2.Right_of_untils have all
6 |             their bits unset (zero),
7 |           Node2.Old = Old  $\cup$  {form},
8 |           Node2.Next = Next);
9 |     modify This (current node) as follows:
10 |         ( ToBeDone = ToBeDone  $\cup$  (New1(form) \ Old),
11 |           Untils and Right_of_untils have all bits unset,
12 |           Old = Old  $\cup$  {form},
13 |           Next = Next  $\cup$  Next1(form) );
14 |     return Node2;
15 | }
```

Figure 3: The splitting algorithm

Obtaining the automaton

The *States_Set* returned by the expand algorithm represents a graph. Each element it contains represents a state of the automaton. We obtain the transitions of the automaton from the information contained in each object of type *State* (see Section 3.2.1). A *State* has a set of transitions which, starting from their *Source*, lead to this state – this is the way states are connected in the graph. Each transition also contains information in its labels, about all the predicates that must hold for this transition to be fired. Additionally, it contains the bitmap *Accepting*, which determines which accepting sets this transition belongs to.

More precisely, each position in the bitmap corresponds to one accepting set. A transition belongs to those accepting sets for which the bit at their corresponding position in the *Accepting* bitmap is set.

This way we obtain a transition-based generalized Büchi automaton, which is subsequently optimized and degeneralized, as described in Section 3.3.1

3.2.3 Optimizations

At the end of the translation we obtain a transition-based generalized Büchi automaton. On this automaton it is possible to apply optimizations, which can reduce the size of the automaton or the size

of the its accepting sets. For most of the algorithms that follow it is necessary to define the transition graph associated with the transition-based generalized Büchi automaton.

Definition 14 (transition graph). Given a transition-based generalized Büchi automaton $TGB = \langle S, A, T, q_0, F \rangle$, we define its transition graph as a directed graph G such that:

$$G = \langle V, E \rangle$$

where:

$$V = S$$

$$\forall q_i \in V, \forall q_j \in V, (q_i, q_j) \in E \Leftrightarrow \exists a \in A \mid (q_i, a, q_j) \in T$$

We call “SCC-optimizations” a family of optimizations that are based on computing the strongly-connected components of the automaton graph. These are based on optimizations defined by [4, 6], but are applied to transition-based generalized Büchi automata.

A very simple, but very useful optimization that we also perform is to ignore sets of accepting sets that are supersets of other accepting sets. We call this the “superset reduction”. It is clear that, if accepting set F_i is a superset of F_j , (i.e., $F_i \supseteq F_j$), then every execution that contains infinitely often at least one element of F_j will also contain at least one element of F_i . Therefore, the accepting set F_i is redundant and can be eliminated. As mentioned, the number of states of degeneralizer automata depends on the number of sets of accepting sets in the generalized automaton. Since the degeneralized automaton is obtained by computing the product of the generalized one with the degeneralizer, its size will typically be smaller if the degeneralizer is smaller. As a result, it is beneficial to reduce the number of sets of accepting states.

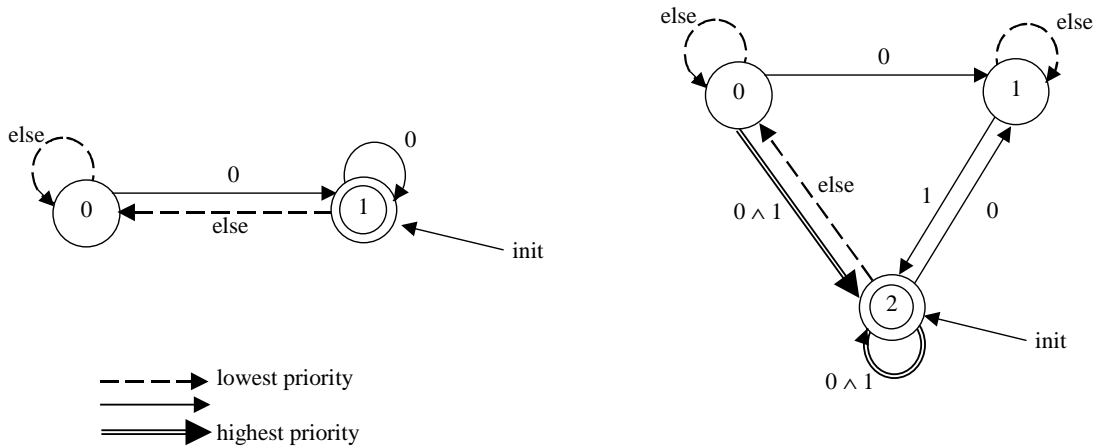


Figure 4. On the left/right, we illustrate the degeneralizer used for a generalized automaton with one/two accepting sets, respectively.

3.3 TGBA2BA

3.3.1 Degeneralization

The automaton generated during stage 2 is turned into a simple Büchi automaton. This is performed by computing the synchronous product between the GBA and a *deterministic* Büchi automaton (we call it the degeneralizer) that expresses the fact that a path can only be accepting if it contains infinitely often at least one accepting transition from each one of the accepting sets [7]. Such

algorithms are well known in the literature, but for completeness, we include here the way in which we generate and use degeneralizers in our LTL2BUCHI.

A degeneralizer for a TGBA is a Büchi automaton with accepting *states*. Degeneralizers have a fixed number of states for a fixed number of accepting sets in their targeted TGBAs. The transitions of a degeneralizer DG are labeled with predicates that relate to accepting sets. We degeneralize a TGBA by computing its synchronous product with the appropriate degeneralizer. In more detail, a joint transition (t_1, t_2) of a DG with a TGBA can be fired if and only if t_2 belongs to the accepting set that the predicate on t_1 requires.

Figure 4 illustrates the degeneralizers used for automata with one and two accepting sets. The degeneralizers we use are *deterministic*; transitions are explored based on their priority – lower priority transitions are only explored if higher priority ones cannot fire. *Else* transitions are taken when no other outgoing transitions from a state are eligible (they have the lowest priority). In Figure 4, note that a degeneralizer only accepts infinite words that satisfy infinitely often the predicates related to each accepting set.

A degeneralizer for n accepting sets is generated with the following algorithm:

```
Büchi generate (int acc_sets) {
    nnodes = acc_sets + 1; //number of automaton nodes
    last = acc_sets; //last automaton node
    for (int i=0; i < nnodes; i++)
        create automaton state Si;
    for (int i=0; i < last; i++) {
        for (int j=last; j>i; j--) {
            create transition "trans" from Si to Sj;
            for (int k=i; k<j; k++)
                Add label k to trans;
        }
        create looping transition for Si labeled with else;
    }
    //now dealing with last node
    create looping transition trans for Slast;
    for (int i=0; i<last; i++)
        add label i to trans;

    for (int i=last-1; i>=0; i--) {
        if (i == 0) then
            create transition from Slast to Si labeled else;
        else {
            create transition trans from Slast to Si;
            for (int j=0; j<i; j++)
                add label j to trans;
        }
    }
}
```

The algorithm above generates labels in the order in which they should be explored, i.e. it generates higher priority transitions first. Note that transitions that have more requirements have higher priority, with “else” transitions having the lowest.

As mentioned, a TGBA is degeneralized by computing its synchronous product with the appropriate degeneralizer, and removing all accepting-related transition labels from the result. In this process, TGBA is considered as a simple Büchi automaton, where all its states are accepting, so the

accepting states of the product are the ones where the degeneralizer is in an accepting state. Figure 5 illustrates the result of degeneralizing the GBA generated for formula $\llbracket a \rrbracket$.

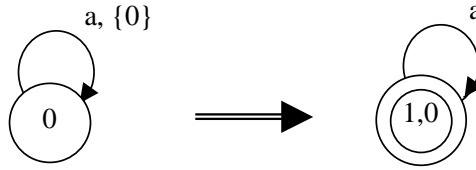


Figure 5. The GBA on the left was obtained by translating formula $\llbracket a \rrbracket$. The automaton on the right was obtained by degeneralizing the GBA using the automaton on the left of Figure 4.

3.3.2 Optimizations

Our Java tool also implements the following further optimizations for reducing the size of the final automaton used for verification:

- strongly-connected component optimizations [4, 6];
- bisimulation reduction;
- fair simulation reduction [4].

4 Correctness Argument

We do not provide a full proof of our algorithm in this report. Rather, we discuss how a proof could be obtained by comparing our procedure to other approaches proven correct.

In essence, the core of our algorithm consists of applying the algorithm of [9] to generate transition-based generalized Büchi automata. There is therefore a direct correspondence between the automata we generate and those generated by the [9] algorithm. One could move between equivalent automata of each type by just applying simple transformations, if one assumes that the information about the *Next* fields of nodes is maintained in states of these automata. All that our algorithm does is that, by moving labels to transitions, it can always merge states that have the same future (same *Next* fields).

The way we compute the accepting sets to which states/transitions belong is based on the way they are computed in [8]. The idea there is to determine, for each *U* formula that has been processed in a *Node*, whether its right-hand side formula has also been processed. In other words, whether the eventualities promised are fulfilled. However, similarly to [9], we do not process some redundant formulae. We therefore need to decide how to treat redundant formulae that are **U**-formulae or right-hand formulae of **U**-formulae. From the definitions of Section 3.2.2, for a **U**-formula to be redundant, its right-hand formula must be syntactically implied by the current *Node*. According to [9], this *Node* is then accepting with respect to the set that the **U**-formula defines. For this reason, when a **U**-formula is redundant, we do not set its bit in the *Utils* bitmap.

On the other hand, it is possible for the right-hand side of a **U**-formula to be redundant, but for its corresponding **U**-formula to have been processed in the *Node* that is being expanded. To cover for this possibility, we always update the *Right_of_Utils* bitmap when the right-hand side of a **U**-formula is encountered.

Our scheme for computing accepting transitions thus avoids re-computing syntactic implications in order to establish accepting states/transitions as is performed in [9], but also avoids storing explicitly all relevant formulae that are processed, as performed in [8].

Transition-based generalized Büchi automata are also used by [7] – such automata are known to be as expressive as state-based Büchi automata. Our degeneralization procedure is also inspired by the

procedure used by [7]. Finally, all other optimizations we apply (fair simulation, superset reduction, simulation, etc) are based on established results.

5 Results and Future Work

The algorithm described in Section 3 has been implemented in Java in the LTL2BUCHI tool, which is being distributed with the JavaPathfinder software from NASA Ames Research Center¹. We have tested the tool with formulae that are typically used in verification, but also formulae that are used to stress test such translators. In all cases, the tool is fast and efficient (in terms of states and transitions of automata generated).

We will not discuss time-efficiency here, since we have not yet set up experiments for fair time comparisons between existing translators. For example, a Web interface was used for some of these [4, 7]. For our translator that runs on fairly fast PCs (Pentium 4 at 1.3 GHz), it never takes more than a few seconds to generate automata even for very complicated formulae, as, for example, formula $\neg(p_1 U (p_2 U (\dots U p_n)\dots))$ for $n=8$, discussed in [7]. However, we have tested the tool for sizes of automata generated. For example, in testing it against “fairness formulae”, on which [7] report that their translation, based on alternating automata, performs better than existing approaches, LTL2BUCHI returns automata with the same number of states. This makes us think that it may not be necessary to implement translations based on Alternating Automata, even though we do not yet have enough evidence to justify this claim. Our tool also achieves better results than the tool by [4] for non-trivial formulae, which shows that fair simulation (implemented by their tool) cannot always compensate for information lost at the earlier stages of the translation.

Of particular interest to us is the comparison with the algorithm by [9], since this is where our algorithm improves the state-of-the art in tableau-based approaches. We perform this comparison by using our implementation of that algorithm in the framework of our tool. With rewriting turned off, we simply compare the generalized (GBA) and degeneralized (BA) automata generated by our algorithm and theirs, before optimizations. Our algorithm returns less states and transitions, considerably so for non-trivial formulae. For example, here are some concrete results:

Formula	LTL2BUCHI (our algorithm)	LTL2AUT ([9])
$((GFa \wedge GFb \wedge GFc) \rightarrow G(d \rightarrow Fe)) \rightarrow G(f \rightarrow Fg)$	GBA: 5 sts 48 trs BA: 11 sts 120 trs	GBA: 28 sts 283 trs BA: 60 sts 635 trs
$\neg(((aUb \vee Ga) \wedge (cUd \vee Gc)) \rightarrow (eUf \vee Ge))$	GBA: 28 sts 123 trs BA: 28 sts 123 trs	GBA: 81 sts 252 trs BA: 81 sts 252 trs
$\neg(aU(bUc))$	GBA: 4 sts 11 trs BA: 4 sts 11 trs	GBA: 8 sts 19 trs BA: 8 sts 19 trs

Such significant reductions that are achieved by our tool in the sizes of both the GBA and of the BA generated make our tool much more efficient. Moreover, for some formulae, “a posteriori” optimizations such as bisimulation and fair simulation do not reduce the automata generated by [9] to the same size as those generated by our algorithm. For example, in the case of the third formula in the above table, the final automaton generated with our approach has 3 states and 7 transitions, whereas the one generated by using the approach of [9] results in 6 states and 12 transitions. Therefore, it is clearly beneficial, not only for time efficiency purposes, but also in terms of the sizes of the final automata obtained, to generate smaller automata as early as possible in the translation process.

¹ please contact dimitra@email.arc.nasa.gov for details.

Although we have not studied sufficiently the relationship between our approach and the one based on Alternating Automata [7], we believe that the great advantage of our optimizations is that, although very efficient, they are based on very simple modifications to the “core” of the tableau-based translation process. It should therefore be easy to update existing tableau-based translators with our algorithm. In the future, we intend to perform more systematic testing and comparison of our approach with existing approaches.

6 References

- [1] Visser, W., Havelund, K., Brat, G., and Park, S. "Model Checking Programs", in *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*. 11-15 September 2000, Grenoble, France. IEEE Computer Society, pp. 3-11. Y. Ledru, P. Alexander, and P. Flener, Eds.
- [2] Holzmann, G.J., *The Model Checker SPIN*. IEEE Transactions on Software Engineering, Vol. **23**(5), May 1997: pp. 279-295.
- [3] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M., *Memory-Efficient Algorithms for the Verification of Temporal Properties*. Formal Methods in System Design, Vol. **1**, 1992: pp. 275-288.
- [4] Etessami, K. and Holzmann, G. "Optimizing Buechi automata", in *Proc. of the 11th International Conference on Concurrency Theory (CONCUR'2000)*. August 2000, Pennsylvania, USA. Springer, LNCS 1877.
- [5] Clarke, E.M., Grumberg, O., and Peled, S.A., *Model Checking*: The MIT press, 1999.
- [6] Somenzi, F. and Bloem, R. "Efficient Buechi automata from LTL Formulae", in *Proc. of the 12th International Conference on Computer Aided Verification (CAV 2000)*. July 2000, Chicago, USA. Springer, LNCS 1855. E.A. Emerson and A.P. Sistla, Eds.
- [7] Gastin, P. and Oddoux, D. "Fast LTL to Buechi Automata Translation", in *Proc. of the 13th International Conference on Computer Aided Verification (CAV 2001)*. July 2001, Paris, France. Springer, LNCS 2102, pp. 53-65. G. Berry, H. Comon, and A. Finkel, Eds.
- [8] Gerth, R., Peled, D., Vardi, M.Y., and Wolper, P. "Simple On-the-fly Automatic Verification of Linear Temporal Logic", in *Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95)*. June 1995, Warsaw, Poland, pp. 3-18.
- [9] Daniele, M., Giunchiglia, F., and Vardi, M.Y. "Improved Automata Generation for Linear Temporal Logic", in *Proc. of the 11th International Conference on Computer Aided Verification (CAV 1999)*. July 1999, Trento, Italy. Springer, LNCS 1633.