SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata

Alexandre Duret-Lutz and Denis Poitrenaud
Laboratoire d'Informatique de Paris 6, Université P. & M. Curie,
4 Place Jussieu, 75252 Paris Cedex 05, France
Alexandre.Duret-Lutz@lip6.fr, Denis.Poitrenaud@lip6.fr

***Abstract*— Spot is a C++ library offering model checking bricks that can be combined and interfaced with third party tools to build a model checker. It relies on Transition-based Generalized Büchi Automata (TGBA) and does not need to degeneralize these automata to check their emptiness. We motivate the choice of TGBA by illustrating a very simple (yet efficient) translation of LTL into TGBA. We then show how it supports on-the-fly computations, and how it can be extended or integrated in other tools.**

I. Introduction

This paper presents Spot (Spot Produces Our Traces), an object-oriented model checking library written in C++. Unlike model checkers, which are programs with a fixed *modus operandi*, the library has no hard-wired operating procedure. It provides a collection of data types and algorithms that can be used as bricks to build a model checker, or experiment extensions.

One striking characteristic of Spot is that it relies on a kind of automata called Transition-based Generalized Büchi Automata (TGBA). These automata encompass traditional Büchi automata, they allow more compact translations of LTL formulæ, and they can be checked for emptiness without being degeneralized. Furthermore, third party state-graph generators or algorithms can be easily plugged into Spot by masquerading as TGBA.

The paper is organized as follows. This introduction presents the automata-theoretic approach and the linear-time temporal logic used by the library. We then present TGBA, and illustrate why they allow shorter automata while translating LTL formulæ. A third section discusses the library itself, and shows how the TGBA interface allows on-the-fly computation and third party extensions. Finally, we review a few such extensions to illustrate this last point.

A. On Model Checkers and the Automata-Theoretic Approach

The automata-theoretic approach to model checking [1] splits the verification process into four operations. These operations are pictured as rounded boxes on Fig. 1, while the square boxes are the data input and output.

1) Computation of the state graph of the model M. This state graph can be seen as an ω-automaton A_M whose language, $\mathscr{L}(A_M)$, is the set of all possible executions of the system.
2) Translation of the temporal property φ into a ω-automaton $A_{\neg\varphi}$ whose language, $\mathscr{L}(A_{\neg\varphi})$, is the set of all executions that would invalidate φ.
3) Synchronized product of these two objects. This constructs an automaton $A_M \otimes A_{\neg\varphi}$ whose language is $\mathscr{L}(A_M) \cap \mathscr{L}(A_{\neg\varphi})$, that is, the set of executions of the model M that invalidate the temporal property φ.
4) Emptiness check of this product. This operation tells whether $A_M \otimes A_{\neg\varphi}$ accepts an infinite word, and can return such a word (a counter-example) if it does. The model M verifies φ iff $\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \emptyset$.

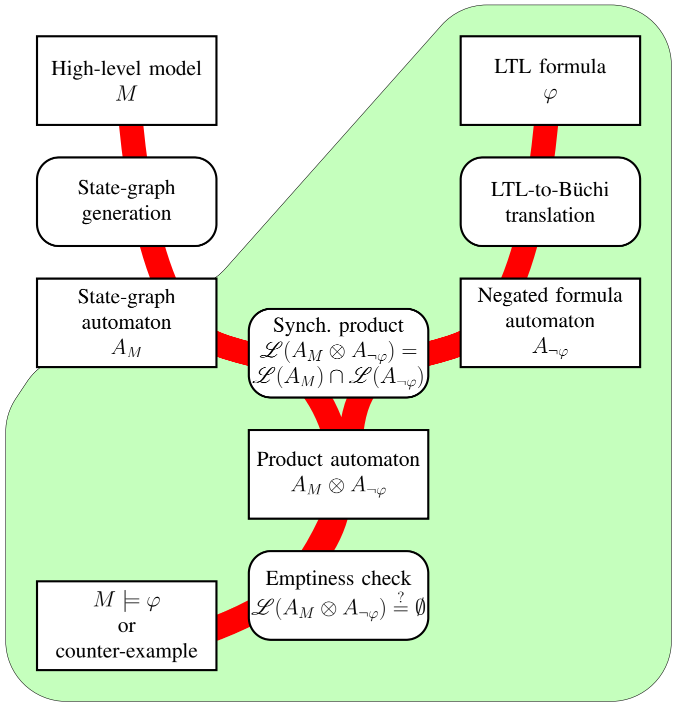

High-level model M
LTL formula φ
State-graph generation
LTL-to-Büchi translation
State-graph automaton A_M
Synch. product $\mathscr{L}(A_M \otimes A_{\neg\varphi}) = \mathscr{L}(A_M) \cap \mathscr{L}(A_{\neg\varphi})$
Negated formula automaton $A_{\neg\varphi}$
Product automaton $A_M \otimes A_{\neg\varphi}$
$M \models \varphi$ or counter-example
Emptiness check $\mathscr{L}(A_M \otimes A_{\neg\varphi}) \overset{?}{=} \emptyset$

Fig. 1. The automata-theoretic approach to LTL model checking.

Although these steps are usually studied separately, in practice they are often tightly tied by the implementation. This is due to traversal optimizations which impede traditional pipeline approaches. One such optimization is *on-the-fly model checking*, where the computation of the product, state graph, and formula automaton are all driven by the progression of the emptiness-check procedure: nothing is computed until it is required. This is an optimization because it may allow the emptiness-check to answer without computing the entire state space; however this hinders reusability and extensibility because the different phases need to be interleaved in the code.

One of Spot's design goal is to implement each step of this automata-theoretic approach independently, so they can be combined or replaced at will by users. However this modularity is achieved in a way that does not preclude on-the-fly computations. We will come to how this is done after we have motivated the choice of automata we use.

The shaded part of Fig. 1 corresponds to algorithms offered by Spot. It should be noted that the state-graph generation has been left out of the library, as it is expected to be carried out by third party tools. We will come back to this in a subsequent section too.

B. Linear-Time Temporal Logic

We consider the future fragment of the propositional linear-time temporal logic, henceforth referred to simply as LTL. LTL formulæ are constructed from a set AP of atomic propositions, the usual boolean operators ($\neg$, $\vee$, $\wedge$) and two temporal operators: X (next) and U (until).

For an infinite word $\sigma = \sigma(0)\sigma(1)\sigma(2)\ldots$ over the alphabet 2^{AP} we denote $\sigma(n) \in 2^{AP}$ its nth letter, and σ^i its suffix starting at letter i (i.e., $\sigma^i(n) = \sigma(i+n)$).

Whether a word σ verifies an LTL formula φ, written $\sigma \models \varphi$, is defined inductively on the formula as follows.

$$\sigma \models p \quad \text{iff } p \in \sigma(0) \tag{1}$$

$$\sigma \models \varphi_1 \wedge \varphi_2 \quad \text{iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \tag{2}$$

$$\sigma \models \varphi_1 \vee \varphi_2 \quad \text{iff } \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \tag{3}$$

$$\sigma \models \neg\varphi \quad \text{iff } \neg(\sigma \models \varphi) \tag{4}$$

$$\sigma \models \mathsf{X}\,\varphi \quad \text{iff } \sigma^1 \models \varphi \tag{5}$$

$$\sigma \models \varphi_1 \,\mathsf{U}\, \varphi_2 \quad \text{iff } \exists i \geqslant 0 \begin{cases} \sigma^i \models \varphi_2 \text{ and} \\ \forall j, 0 \leqslant j < i, \sigma^j \models \varphi_1 \end{cases} \tag{6}$$

II. State vs. Transition-Based Büchi Automata

Büchi automata are ω-automata traditionally used in LTL verification. These automata accept only the infinite words that traverse infinitely often a set of states called the *acceptance set*. Generalized Büchi automata can have multiple acceptance sets that must each be traversed infinitely often.

Properties of the models as well as propositions of the formula are commonly carried by states of the automata. In some LTL-to-automata translations, propositions are carried by transitions, but acceptance conditions are always relative to states.

It was recently recognized that transition-based automata, where acceptance conditions are defined in term of transitions, can lead to translations of LTL formulæ to smaller automata.

More formally, a *Transition-based Generalized Büchi Automata* (TGBA) over the alphabet Σ is a Büchi automaton with labels on transitions, and generalized acceptance conditions on transitions too. It can be defined as a tuple $A = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q^0, \delta \rangle$ where

- Σ is an alphabet,
- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{F}$ is a finite set of *acceptance conditions*,
- $q^0 \in \mathcal{Q}$ is a distinguished initial state,
- $\delta \subseteq \mathcal{Q} \times (2^\Sigma \setminus \{\emptyset\}) \times 2^\mathcal{F} \times \mathcal{Q}$ is the transition relation, where each transition carries a nonempty set of letters of the alphabet and a set of acceptance conditions.

For the purpose of model checking we have AP equal to the set of all atomic proposition that can characterize a configuration, and use these automata with $\Sigma = 2^{AP}$ (i.e., each configuration of the system can be mapped into a letter of Σ).

An infinite word σ over the alphabet Σ is accepted by A if there exists an infinite sequence $\rho = (q_0, l_0, f_0, q_1)(q_1, l_1, f_1, q_2)\ldots$ of transitions of δ, starting at $q_0 = q^0$, and such that $\forall i \geqslant 0, \sigma(i) \in l_i$, and $\forall f \in \mathcal{F}, \forall i \geqslant 0, \exists j \geqslant i, f \in f_j$. That is, each transition matches the corresponding letter of the word, and ρ traverses each acceptance condition infinitely often.

To the best of our knowledge, LTL translators have started using this kind of automaton five years ago [2].[1]

Translation algorithms by Gastin & Oddoux [4], Giannakopoulou & Lerda [5] who actually coined the acronym TGBA, and Thirioux [6] use TGBA only as an intermediate step before they finally produce a non-generalized Büchi automaton (with one acceptance condition on states). Degeneralizing an automaton can induce a linear blowup of the automaton, which implies a bigger product and a slower emptiness check. However this last step is motivated by mainstream emptiness-check algorithms that work only with one acceptance set of states [7].

Other people have provided not only LTL to TGBA translators, but also emptiness check algorithms that work directly on these TGBA, avoiding the aforementioned blowup. This includes work from Couvreur [2], [8] and Tauriainen [9].

Spot implements the two LTL translation algorithms of Couvreur [2], [8] as well as his emptiness check algorithm [2], so it can handle the full Fig. 1 with TGBA. It also has methods to degeneralize a TGBA, so it can feed algorithms designed for more conventional Büchi automata.

A. Translating LTL formulæ into TGBA

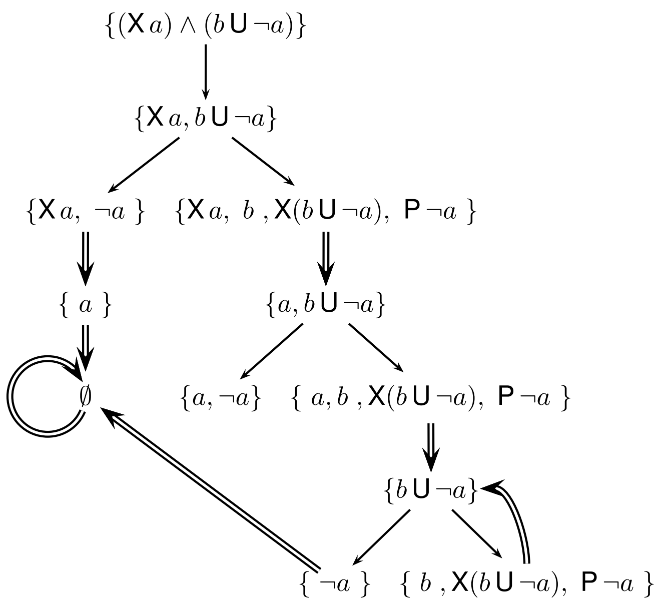
Most LTL to Büchi automata translation algorithms in use today are based on tableau methods for LTL [10].

In propositional logic, a tableau can prove that a formula is satisfiable. It is a tree whose nodes are sets of formulæ. The root is a singleton containing the formula to check, and the children of a node are computed by applying one of the 7 first rewriting rules of Fig. 2. For instance if a node contains $f \wedge g$ we can decide to apply the 5th rule and create two

[1]For history's sake, we should mention Michel's translation [3] which used a cousin structure 15 years earlier. He translates an LTL formula into a network of transducers with labels on transitions, using acceptance conditions that are the opposite of those of TGBA: *unstable graphs* denote sets of transitions from which runs of the transducer are required to exit infinitely often. *Unstable graphs* correspond to the concept of *promises* we will introduce in the sequel.

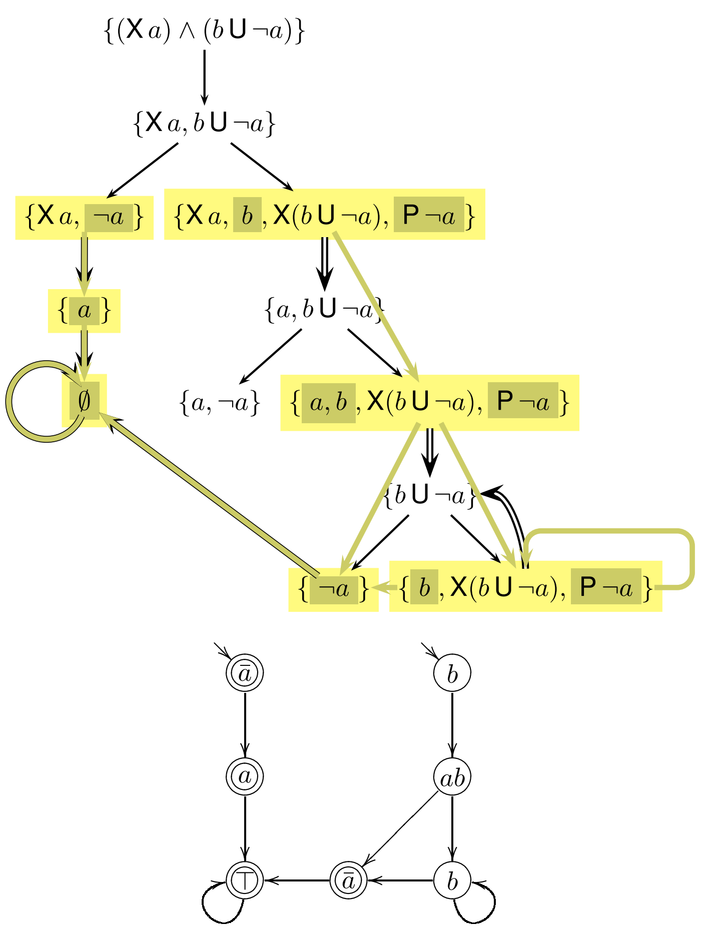
formula	1st child	2nd child
$\{\Gamma, \neg\top\}$	$\{\Gamma, \bot\}$	
$\{\Gamma, \neg\bot\}$	$\{\Gamma, \top\}$	
$\{\Gamma, \neg\neg f\}$	$\{\Gamma, f\}$	
$\{\Gamma, f \wedge g\}$	$\{\Gamma, f, g\}$	
$\{\Gamma, f \vee g\}$	$\{\Gamma, f\}$	$\{\Gamma, g\}$
$\{\Gamma, \neg(f \wedge g)\}$	$\{\Gamma, \neg f\}$	$\{\Gamma, \neg g\}$
$\{\Gamma, \neg(f \vee g)\}$	$\{\Gamma, \neg f, \neg g\}$	
$\{\Gamma, \neg \mathsf{X} f\}$	$\{\Gamma, \mathsf{X} \neg f\}$	
$\{\Gamma, f \mathbin{\mathsf{U}} g\}$	$\{\Gamma, g\}$	$\{\Gamma, f, \mathsf{X}(f \mathbin{\mathsf{U}} g), \mathsf{P}\, g\}$
$\{\Gamma, \neg(f \mathbin{\mathsf{U}} g)\}$	$\{\Gamma, \neg f, \neg g\}$	$\{\Gamma, \neg g, \mathsf{X}\neg(f \mathbin{\mathsf{U}} g)\}$

Fig. 2. Tableau rules for LTL.



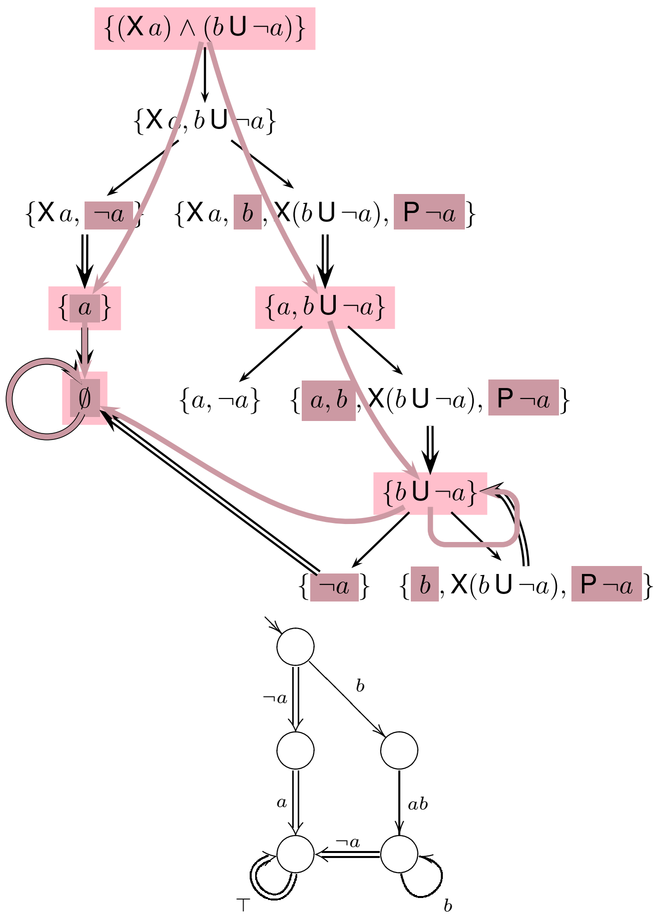
$\{(\mathsf{X}\, a) \wedge (b \mathbin{\mathsf{U}} \neg a)\}$
$\{\mathsf{X}\, a, b \mathbin{\mathsf{U}} \neg a\}$
$\{\mathsf{X}\, a, \neg a\}$
$\{\mathsf{X}\, a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{a\}$
$\{a, b \mathbin{\mathsf{U}} \neg a\}$
$\emptyset$
$\{a, \neg a\}$
$\{a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{b \mathbin{\mathsf{U}} \neg a\}$
$\{\neg a\}$
$\{b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$

Fig. 3. Tableau for $(\mathsf{X}\, a) \wedge (b \mathbin{\mathsf{U}} \neg a)$.



$\{(\mathsf{X}\, a) \wedge (b \mathbin{\mathsf{U}} \neg a)\}$
$\{\mathsf{X}\, a, b \mathbin{\mathsf{U}} \neg a\}$
$\{\mathsf{X}\, a, \neg a\}$
$\{\mathsf{X}\, a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{a\}$
$\{a, b \mathbin{\mathsf{U}} \neg a\}$
$\emptyset$
$\{a, \neg a\}$
$\{a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{b \mathbin{\mathsf{U}} \neg a\}$
$\{\neg a\}$
$\{b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\bar{a}$
b
a
ab
$\top$
$\bar{a}$
b

Fig. 4. Using leaves to build a state-based generalized Büchi automaton.



$\{(\mathsf{X}\, a) \wedge (b \mathbin{\mathsf{U}} \neg a)\}$
$\{\mathsf{X}\, a, b \mathbin{\mathsf{U}} \neg a\}$
$\{\mathsf{X}\, a, \neg a\}$
$\{\mathsf{X}\, a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{a\}$
$\{a, b \mathbin{\mathsf{U}} \neg a\}$
$\emptyset$
$\{a, \neg a\}$
$\{a, b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\{b \mathbin{\mathsf{U}} \neg a\}$
$\{\neg a\}$
$\{b, \mathsf{X}(b \mathbin{\mathsf{U}} \neg a), \mathsf{P}\neg a\}$
$\neg a$
b
a
ab
$\neg a$
$\top$
b

Fig. 5. Using roots to build a transition-based generalized Büchi automaton.

children: one labeled with f and the other unrewritten terms, the other labeled with g and these same unrewritten terms. This process is repeated on these new leaves until no more rules can be applied. As can be seen from the rules, this method essentially is a transformation to disjunctive normal form. At the end, leaves (that represent conjunctions) can contain only propositions and their negations, as well at $\top$ (true) and $\bot$ (false). A branch is said to be closed if its leaf contains $\bot$ or both proposition and its negation, and the root formula is satisfiable if the tableau has a non-closed branch.

Tableau methods for LTL work similarly, but additionally have to deal with the temporal aspect of LTL formulæ. They proceed by slicing the tableau in different instants. The tableau rules are augmented with three new rules, on the bottom of Fig. 2, based on the following equivalences:

$$\neg \mathsf{X} f \equiv \mathsf{X} \neg f \tag{7}$$

$$f \mathbin{\mathsf{U}} g \equiv g \lor (f \land \mathsf{X}(f \mathbin{\mathsf{U}} g)) \tag{8}$$

$$\neg(f \mathbin{\mathsf{U}} g) \equiv \neg g \land (\neg f \lor \mathsf{X} \neg(f \mathbin{\mathsf{U}} g)) \tag{9}$$

These equivalences (which can be proven from equations (4) to (6)) isolate what has to be verified at one instant from what has to be verified next. Using these rules it is possible to develop a tableau whose leaves contain only $\top$, $\bot$, atomic propositions, negated atomic propositions, and X-prefixed LTL formulæ. Such a tableau can be seen in the top four nodes of Fig. 3. (We will discuss $\mathsf{P}\, g$ soon.)

This four-node tableau can be understood as follows: to verify the root formula $(\mathsf{X}\, a) \land (b \mathbin{\mathsf{U}} \neg a)$ one must verify either the first leaf, $(\mathsf{X}\, a) \land \neg a$, or the second, $(\mathsf{X}\, a) \land b \land \mathsf{X}(b \mathbin{\mathsf{U}} \neg a)$.

Once the tableau for an instant has been fully developed and no rewriting rule can be applied anymore, it can be extended to take the next instant into account. This is done by starting a new tableau from each leaf, using the set of X-prefixed formulæ as root. The double arrows on figure 3 represent these jumps to the next instant.

Because (8) and (9) are recursive equations, we need to identify nodes of the tableau that share the same label, so that the tableau is a finite graph instead of being an infinite tree.

At this point it should be fairly intuitive to the reader why an automata labeled with the leaves at the different instants of the tableau will recognize (at least) the sequences that verify the formula. The reasoning is that if we have built an automaton for $(\mathsf{X}\, a) \land (b \mathbin{\mathsf{U}} \neg a)$, then that automaton should accept either words that start with a letter that verifies $\neg a$ and then verify a, or words whose first letter verifies b and that then verify $a \land (b \mathbin{\mathsf{U}} \neg a)$. Fig. 4 and Fig. 5 show how this tableau can be used to construct an automaton with labels respectively on states and transitions.

The only hard part of tableau methods is how to avoid the unwanted infinite loops induced by (8). Indeed the $f \mathbin{\mathsf{U}} g$ formula guarantees that g will be verified eventually (see (6)), but when applying the tableau rule corresponding to (8) we construct a loop in which f could be verified infinitely often without g being ever verified. Such a loop can be seen at the bottom right of our example tableau, where we could loop over b infinitely often.

The way to catch these unwanted loop has been a source of several variants of tableau methods.

One solution is to introduce a new operator into the tableau, P, to express *promises*. When $f \mathbin{\mathsf{U}} g$ has to be verified, either (i) g holds, or (ii) f holds, we should verify $f \mathbin{\mathsf{U}} g$ next, and we also promise to verify g eventually ($\mathsf{P}\, g$). Promises are easy to express in Büchi automata because they are the opposite of acceptance conditions. So each promise in the tableau will be mapped onto an acceptance condition of the automaton, carried by all transitions or states (depending on the type of automata we construct) that do not make this promise. This guarantees that a sequence of the automata cannot postpone a promised formula indefinitely. In Fig. 4 and 5 there is only one promise ($\mathsf{P} \neg a$), so the single acceptance set of the automata has been represented by double circles or double arrows.

These figures exhibit why a transition-based Büchi automaton constructed using this method will always be smaller than a state-based one: a tableau has always less roots than leaves...[2]

In practice, one would not implement tableau methods as presented. The first 7 tableaux rules, whose purpose is to construct a disjunctive normal form, are better replaced by any smarter algorithm that minimizes the number of leaves. Also, leaves that have an identical set of X-formulæ and P-formulæ can be merged; meaning a transition (or state) of the resulting automata will be labeled by a propositional formula rather than by a propositional conjunction.

Since we are only interested in showing why TGBA are interesting we do not provide a proof for this tableau construction. We refer the interested reader to the proof of Couvreur's translation [2]: what he calls $a_{f \mathbin{\mathsf{U}} g}$ is what we write $\mathsf{P}\, g$.

B. LTL Translation in Spot

The above tableau construction can be efficiently implemented using BDD as presented by Couvreur [2]. Spot implements this algorithm, augmented with techniques presented by Sebastiani & Tonetta [11] to create more deterministic automata, LTL formula simplifications introduced by Etessami & Holzmann [12] and Somenzi & Bloem [13], and automaton simplification by Etessami et al. [14]. This first algorithm is referred to as Spot/FM.

Spot also implements a second algorithm also presented by Couvreur [8], that we will call Spot/LaCIM. This algorithm is not intended to produce small automata: it constructs automata that can be represented compactly using BDDs. The representation is compact and efficient to use, but the automaton it represents can be big.

Many LTL translation algorithms have been developed over the last 20 years. Most of them strive to produce small automata in an attempt to produce a smaller synchronized product (see Fig. 1) and hence to speed up the emptiness check. Another way to reduce the product is to output more

[2]Roots and leaves were respectively designated as *pre-states* and *states* by Wolper [10]. While this naming is suitable for classical Büchi automata (*states* of the tableau corresponds to states of the automaton), it can be confusing when building TGBA (*pre-states* become states of the automaton).

deterministic automata; this is done in Modella and Spot/FM. As already said, not degeneralizing the formula automaton will also help keeping the product small, this is why Spot promotes TGBA.

TABLE I
TRANSLATING 39 LTL FORMULÆ.

Algorithm reference and implementation		Automata st.	Automata tr.	Products st.	Products tr.
Translation to degeneralized Büchi automata:					
[2]	Spot/FM	181	453	36139	1380196
[12]	Spin 4.0.6	277	1196	54200	3548935
[15]	LBT 1.2.1	844	5584	161040	12530060
[13]	Wring 1.1.0	407	5339	58762	4377735
[4]	LTL2BA 1.0	205	624	40200	2480261
[16]	LTL→NBA	235	1022	40800	2777045
[11]	Modella 1.5.1	561	1181	112093	4078015
Translation to TGBA:					
[2]	Spot/FM	153	403	30562	1163346
[8]	Spot/LaCIM	494	4379	98800	5465590
[4]	LTL2BA 1.0 (unfair)	169	506	33200	1941344

Table I shows how the Spot translations compete with other freely available implementations. We used LBTT 1.1.2 [17] to run several LTL translators on 39 classic LTL formulæ, and compute their synchronized product with a random state-graph. The table shows the cumulated size in states and transitions of these automata and products. The formulæ used are the 27 formulæ listed by Somenzi & Bloem [13], and the 12 from Etessami & Holzmann [12].

The first part of the table shows the size for degeneralized Büchi automata; since these are the most used automata (e.g., Spin can be fed with such automata directly). Although Spot/FM, LTL2BA, and LTL→NBA usually produce comparable automata, those of Spot are usually more deterministic: it can be seen that the product is smaller, especially its number of transitions.

The second part of the table shows the size of TGBA for the same formulæ. We have extracted the intermediate TGBA used by LTL2BA to produce a degeneralized automata to have a comparison, but this is quite unfair because this skips all the post simplifications of LTL2BA.

III. TGBA IN SPOT

TGBA are handled in Spot using an abstract class, `tgba`, with a minimalist interface. Baring other technical details, this interface boils down to two functions: `get_init_state()` returns the initial state, and `succ_iter()` returns an iterator over the successors of a state. This iterator will also supply transition labels (properties, acceptance sets).

Several implementations of this abstract `tgba` are offered by Spot. For instance an automaton can be encoded either explicitly, or using a BDD relation. `tgba` is also a means of allowing on-the-fly computation and third party extensions.

A. `tgba` to Support On-The-Fly Computations

An algorithm that produces a TGBA, for instance a synchronized product of two automata, can be implemented as a subclass of `tgba`. An object of this class represents the result of the algorithm, although it has not been computed yet. Only when `get_init_state()` or `succ_iter()` are called does any computation actually take place.

Implementing on-the-fly algorithms this way allows us to combine several on-the-fly computations. Nesting products is such a combination, and it is particularly useful when composing models or formulæ.

Thanks to this abstraction, on-the-fly computation does not incur any loss of modularity: all algorithms are well decoupled and can be worked on separately.

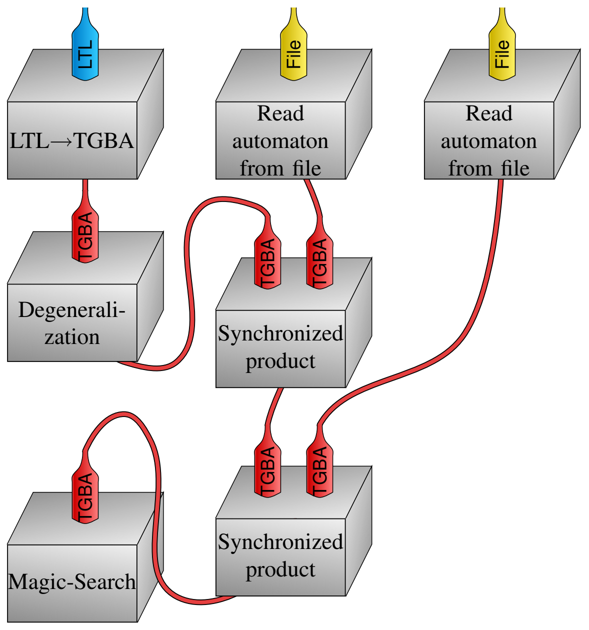


Fig. 6. Composing model checking bricks.

Fig. 6 shows a contrived example of such modularity in on-the-fly checking. This setup has three inputs: one LTL property and two files containing TGBA automata. The LTL is first translated into a TGBA, then this automata is degeneralized and synchronized successively with the two automata, and finally checked for emptiness using the magic-search [18]. In this setup, the degeneralization and the two products are computed on-the-fly, following the progression of the magic-search. Although this is not done in Spot, it should not be a problem to translate the LTL formula and read the automata on-the-fly too.

B. `tgba` as an Interface for Third Party Extensions

The above example was artificial, because automata are not usually read from files in practice. Rather, one would like to generate a state-graph on-the-fly from a high-level formalism such as Petri Net, or Promela. Since Spot does not support any such formalism, this generation of the state graph has to be done outside the library.

Fortunately, the `tgba` interface can also be seen as a standardized way to extend Spot. Users can implement their own subclasses of `tgba` and use them with Spot's algorithms transparently.

Fig. 7 illustrates how we have connected Spot to GreatSPN[3]. GreatSPN can produce a symbolic reachability graph (SRG) for a Colored Petri net by exploiting its symmetries [19]. We have implemented this interface as a `tgba` subclass whose methods simply delegate their work to the corresponding procedures of GreatSPN. From the point of view of Spot, an SRG appears as any other TGBA.

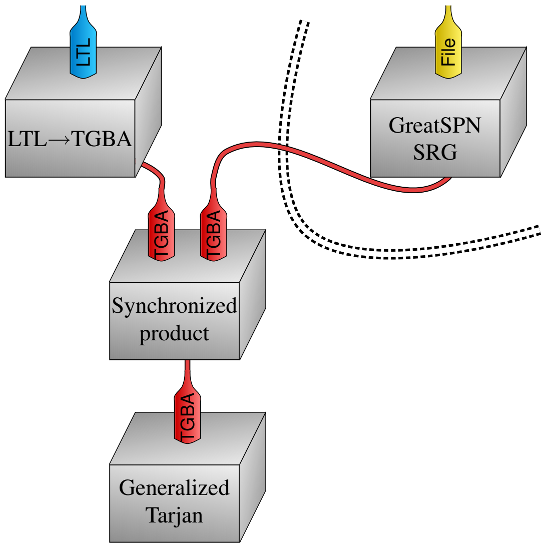


Fig. 7. Interfacing third party state graph generators.

Fig. 7 also shows a concrete implementation of the automata-theoretic approach to model checking depicted in Fig. 1. The *Generalized Tarjan* algorithm corresponds to the algorithm presented by Couvreur [2].

Roughly speaking, SRG exploits global symmetries of the system and regroups items of the system that are not distinguished by the formula [20].

Another approach, worked out by Baarir et al. [21] does not consider the formula as a whole, but will rather consider the automaton for this formula, and computes the equivalence classes induced by a transition of this automaton during the synchronization. This second approach is called the Symbolic Synchronized Product (SSP), and has been connected to Spot as shown in Fig. 8.

As can be seen on that figure, the integration goes beyond simply plugging a state graph generator. We are here using a third party synchronized product.

C. `tgba` as an Input Formalism

It seems important to us that our interface be based on a low-level formalism like TGBA.

Model checkers usually define their own input formalism, that preserves properties which are important to the tool. When it comes to integration with other tools, models have to be converted between the various formalisms used. This turns out to be difficult when the features of each formalism are disjoint, and results in the definition of a core formalism which is the intersection of all the others.

By allowing input to be given directly in our low-level formalism (viz., TGBA) we hope to allow many high-level formalisms to be used. For instance the GreatSPN interface is a way to convert Well-Formed Petri Nets into TGBA.

Note that this is also true for formulæ. The automata-theoretic approach can be used to check any property that can be expressed as a TGBA. Translating an LTL formula is only one way to produce such a TGBA.

We should also emphasize that TGBA can be seen as a superclass of the other Büchi automata. A Büchi automaton with labels on states (or labels on transitions but acceptance conditions on states) can be rewritten as TGBA of equal size easily (and on-the-fly), while the converse is not true. This means that Spot can reuse any algorithm that produces a Büchi automaton, but cannot otherwise directly use algorithms that expect such an automaton as input, without prior conversion.

[3]`http://www.di.unito.it/~greatspn/`

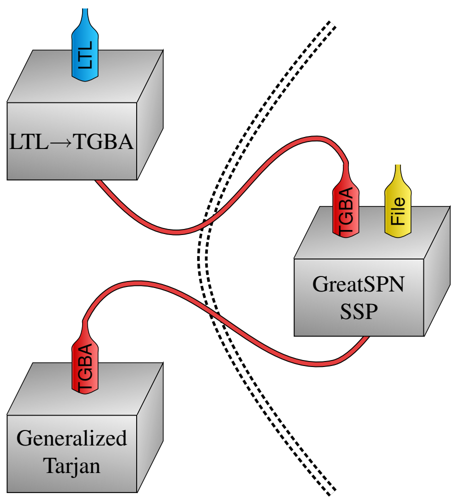


Fig. 8. Interfacing third party synchronized product.

IV. Other uses of Spot

The above couplings between Spot and GreatSPN to implement SRG and SSP have been successfully used to model check a medium-sized model: the heart of the PolyORB middleware, modeled as a Well-Formed Petri Net [22]. These results have yet to be published, but they are the first realistic uses of our tools.

Besides SRG and SSP, Spot is also used to drive a few other state-graph generator methods developed in our team. An interface with the symbolic method of Haddad et al. [23] already exists, and one to the symbolic symbolic state space representation of Thierry-Mieg et al. [24] is being developed.

Finally, Spot comes with a set of Python bindings that allows the library to be used from Python scripts (often more convenient than C++ when experimenting). These bindings were for instance used to implement our on-line LTL2TGBA translator.

V. Conclusion

Our contribution in this paper is twofold. We have presented an easy-to-teach explanation of tableau methods to translate LTL formulæ into generalized Büchi automata with state or transitions labels. This presentation introduces promises, which—to our knowledge—is new.

We have also introduced a model checking library, Spot, that offers a set of model checking bricks supporting on-the-fly computation and that can be easily extended. Spot uses TGBA, which encompass existing automata and allows translation of LTL to smaller automata. Besides translating LTL formulæ, Spot offers algorithms to simplify LTL formulæ, simplify TGBA, build synchronized products, as well as algorithms to perform emptiness check and find counter-examples.

Spot is distributed as free software at `http://spot.lip6.fr/`. Our hope is that it can serve the community as a ground to experiment and develop new model checking algorithms.

VI. Acknowledgments

The Spot library has been implemented by Alexandre Duret-Lutz, Thomas Martinez, Denis Poitrenaud, and Rachid Rebiha, with suggestions from Jean-Michel Couvreur and Jean-Michel Ilié. The interface with GreatSPN for SRG and SSP have been done with the help of Yann Thierry-Mieg and Soheib Baarir. Kais Klai implemented the interface to the state-graph generator he developed during his PhD [23], [25], and Philippe Garilli is working on the integration of the state-graph generator of Thierry-Mieg et al. [24].

References

[1] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Proc. of the 8th Banff Higher Order Workshop*, ser. LNCS, F. Moller and G. M. Birtwistle, Eds., vol. 1043. Banff, Alberta, Canada: Springer-Verlag, 1996, pp. 238–266.

[2] J.-M. Couvreur, "On-the-fly verification of temporal logic," in *Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, ser. LNCS, J. M. Wing, J. Woodcock, and J. Davies, Eds., vol. 1708. Toulouse, France: Springer-Verlag, September 1999, pp. 253–271.

[3] M. Michel, "Algèbre de machines et logique temporelle," in *Proc. of the Symp. on Theoretical Aspects of Computer Science (STACS'84)*, ser. LNCS, M. Fontet and K. Mehlhorn, Eds., vol. 166, Paris, France, April 1984, pp. 287–298.

[4] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *Proc. of the 13th Int. Conf. on Computer Aided Verification (CAV'01)*, ser. LNCS, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer-Verlag, 2001, pp. 53–65.

[5] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of LTL formulae to Büchi automata," in *Proc. of the 22nd Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, ser. LNCS, D. Peled and M. Vardi, Eds., vol. 2529. Houston, Texas: Springer-Verlag, November 2002, pp. 308–326.

[6] X. Thirioux, "Simple and efficient translation from LTL formulas to Büchi automata," in *Proc. of the 7th Int. ERCIM Workshop in Formal Methods for Industrial Critical Systems (FMICS'02)*, ser. ENTCS, R. Cleaveland and H. Garavel, Eds., vol. 66(2). Málaga, Spain: Elsevier, July 2002.

[7] G. J. Holzmann, D. A. Peled, and M. Yannakakis, "On nested depth first search," in *Proc. of the 2nd Spin Workshop*, ser. DIMACS, J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, Eds., vol. 32. AMS, May 1996.

[8] J.-M. Couvreur, "Un point de vue symbolique sur la logique temporelle linéaire," in *Actes du Colloque LaCIM 2000*, ser. Publications du LaCIM, P. Leroux, Ed., vol. 27. Montréal: Université du Québec à Montréal, August 2000, pp. 131–140.

[9] H. Tauriainen, "On translating linear temporal logic into alternating and nondeterministic automata," Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Research Report A83, December 2003.

[10] P. Wolper, "The tableau method for temporal logic: An overview," *Logique et Analyse*, no. 110–111, pp. 119–136, 1985.

[11] R. Sebastiani and S. Tonetta, ""more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking," in *Proc. of the 12th Advanced Research Working Conf. on Correct Hardware Design and Verification Methods (CHARME'03)*, ser. LNCS, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds., vol. 2860. L'Aquila, Italy: Springer-Verlag, October 2003, pp. 126–140.

[12] K. Etessami and G. J. Holzmann, "Optimizing Büchi automata," in *Proc. of the 11th Int. Conf. on Concurrency Theory (CONCUR'2000)*, ser. LNCS, C. Palamidessi, Ed., vol. 1877. Pennsylvania, USA: Springer-Verlag, 2000, pp. 153–167.

[13] F. Somenzi and R. Bloem, "Efficient Büchi automata for LTL formulae," in *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'00)*, ser. LNCS, vol. 1855. Chicago, Illinois, USA: Springer-Verlag, 2000, pp. 247–263.

[14] K. Etessami, R. Schuller, and T. Wilke, "Fair simulation relations, parity games, and state space reduction for Büchi automata," in *Proc. of the 28th international collquium on Automata, Languages and Programming*, ser. LNCS, F. Orejas, P. G. Spirakis, and J. van Leeuwen, Eds., vol. 2076. Crete, Greece: Springer-Verlag, July 2001, pp. 694–707.

[15] M. Rönkkö, "LBT: LTL to Büchi conversion," http://www.tcs.hut.fi/Software/maria/tools/lbt/, 1999, implements [26].

[16] C. Fritz, "Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata," in *Proc. of the 8th Int. Conf. on Implementation and Application of Automata (CIAA'03)*, ser. LNCS, O. H. Ibarra and Z. Dang, Eds., vol. 2759. Santa Barbara, California: Springer-Verlag, July 2003, pp. 35–48.

[17] H. Tauriainen, "A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata." in *Proc. of the Concurrency, Specification and Programming 1999 Workshop (CS&P'99)*, H.-D. Burkhard, L. Czaja, H.-S. Nguyen, and P. Starke, Eds., Warsaw, Poland, September 1999, pp. 251–262.

[18] P. Godefroid and G. J. Holzmann, "On the verification of temporal properties," in *Proc. of the 13th Int. Symp. on Protocol Specification, Testing, and Verification (PSTV'93)*, ser. IFIP Transactions, A. A. S. Danthine, G. Leduc, and P. Wolper, Eds., vol. C-16. Liege, Belgium: North-Holland, May 1993, pp. 109–124.

[19] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "A symbolic reachability graph for coloured Petri nets," *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 39–65, April 1997.

[20] Y. Thierry-Mieg, C. Dutheillet, and I. Mounier, "Automatic symmetry detection in well-formed nets," in *Proc. of the 24th Int. Conf. on Application and Theory of Petri Nets (ICATPN'03)*, ser. LNCS, W. van der Aalst and E. Best, Eds., vol. 2679. Eindhoven, The Netherlands: Springer-Verlag, June 2003, pp. 82–101.

[21] S. Baarir, S. Haddad, and J.-M. Ilié, "Exploiting partial symmetries in well-formed nets for the reachability and the linear time model checking problems," in *Proc. of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, September 2004, in press.

[22] J. Hugues, L. Pautet, and F. Kordon, "Refining middleware functions for verification purpose," in *Proc. of the Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, Chicago, Illinois, USA, September 2003.

[23] S. Haddad, J.-M. Ilié, and K. Klai, "Design and evaluation of a symbolic and abstraction-based model checker," in *Proc. of the 2nd Int. Symp. on Automated Technology for Verification and Analysis (ATVA'04)*, National Taiwan University, Taiwan, October 2004, to appear.

[24] Y. Thierry-Mieg, J.-M. Ilié, and D. Poitrenaud, "A symbolic symbolic state space representation," in *Proc. of the 24th Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, Madrid, Spain, September 2004, in press.

[25] K. Klai, "Réseaux de petri: vérification symbolique et modulaire," Ph.D. dissertation, Université Paris 6, France, 2003.

[26] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Proc. of the 15th Workshop on Protocol Specification Testing and Verification (PSTV'95)*. Warsaw, Poland: Chapman & Hall, 1996, pp. 3–18.