Translation of an extended LTL into TBGA in Spot

Damien Lefortier

Technical Report n^o0807, June 2009
revision 2047

Spot is a model checking library centered around the automata approach, and can be used to verify properties expressed using LTL (Linear Temporal Logic) formulæ on models represented as TGBA (Transition-based Generalized Büchi automata). The library offers two translation algorithms from LTL formulæ to TGBA, one of the main stages of the approach. We present an extension of one of these algorithms to an extended LTL where operators are represented by finite automata, allowing Spot to verify properties that were not expressible before. We also present how we could add some features of PSL (Property Specification Language) in our extension.

Spot est une bibliothèque de *model checking* qui permet de vérifier des propriétés exprimées en logique temporelle à temps linéaire (LTL) sur des modèles représentés par des automates de Büchi généralisés basés sur les transitions (TGBA). Spot propose actuellement deux algorithmes de traduction de LTL en TGBA, une des étapes principales de l'approche automate. Nous présentons une nouvelle traduction en TGBA d'une LTL étendue dont les opérateurs sont représentés par des automates finis, permettant ainsi à Spot de vérifier des propriétés qui n'étaient pas exprimables auparavant. Nous présenterons aussi de quelles façons nous pourrions ajouter certaines fonctionnalités de PSL (Property Specification Language) à notre extension.

Keywords
extended linear temporal logic, ELTL, Büchi automaton, LaCIM algortihm, PSL

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
dam@lrde.epita.fr – http://www.lrde.epita.fr

Copying this document

Contents

Introduction **4**

1 Preliminaries **6**
1.1 Linear Temporal Logics 6
1.1.1 Definitions 6
1.1.2 LTL 7
1.1.3 ETL 7
1.2 TGBA 9
1.2.1 Definition 9
1.2.2 Symbolic representation 9

2 LaCIM Algorithm **11**
2.1 LTL 11
2.2 ELTL 13

3 Implementation **15**
3.1 AST 15
3.2 Parser 15
3.3 LaCIM 17

4 ELTL & PSL **19**
4.1 SERE operators 19
4.2 Integration in ELTL 20

Conclusion **22**

Bibliography **23**

Introduction

Model checking is a formal method for verifying finite-state concurrent systems. Typically, the user provides a high-level representation of the system (the model) and the specification to be checked. Specifications about the system are expressed as temporal logic formulæ, and efficient algorithms are used to traverse the model and check if the specification holds or not. Compared to traditional approaches based on simulation, testing, or deductive reasoning, model checking has a number of advantages. In particular, it is automatic and produces a counter-example in case of failure.

The Spot (Duret-Lutz and Poitrenaud, 2004) project aims at producing a flexible and efficient model checking library centered around the automata theoretic approach (Vardi, 1986). In this approach, an execution of the system is seen as an infinite word, where letters are successive configurations of the system. Thus, the whole set of possible executions is seen as the set of infinite words recognized by a finite automaton on infinite words (also called ω-automaton). In this context, the verification is split as follow:

- The high-level model is translated into a state graph automaton whose language is the set of all possible execution of the system modeled. (This step is not offered by Spot.)
- The formula to verify is also translated into an automaton whose language is the set of all executions that would invalidate the formula. So this is actually the automaton for the negated formula.
- The above two automata are combined, using a synchronized product, to create a third automaton whose language is the intersection of the languages of the two automata.
- A last operation, called *emptiness check*, tells whether the language of the synchronized product is empty. If it is not, it contains a sequence of execution that is allowed by the model, but that does invalidate the formula: this is a counter-example. Otherwise the formula holds for all possible execution of the modeled system.

Usually, Büchi automata, which define exactly the ω-regular languages, are used. In Spot we use a special kind of ω-automata called Transition-based Generalized Büchi Automata (TGBA) and a Linear Temporal Logic (LTL) formula is used to specify properties about the system. Unfortunately, these two formalisms are not equivalent, and thus we cannot verify as many properties as the model can hold. In fact, even the simple assertion that "the proposition p holds at least in every other state on a path" is not expressible in LTL (Wolper et al., 1983), but can be represented by the ω-regular expression $(p\top)^\omega$.

Vardi and Wolper (1994) investigated extensions of LTL able to express all ω-regular languages, and we present formally one of these extensions and explain how we integrated it in Spot, including an adaptation of the LaCIM translation algorithm to TGBA for this extended LTL (the second stage of the approach given before) based on Couvreur (2000). This allows Spot to verify properties that were not expressible before.

Outline This paper is organized as follows. In Chapter 1, we introduce formally LTL and the extended LTL (ELTL) we use, also provide an extensive definition of TGBA and how they can be reprensented symbolically – which is going to be useful in Chapter 2 where the LaCIM translation algorithm is presented. Chapter 3 summarizes what has been done for Spot to support ELTL formulæ, including benchmarks of our implementation of LaCIM for ELTL, and finally Chapter 4 presents some interesting features of PSL that could be integrated in our ELTL.

Everything that will be presented is available in the latest development version of Spot which can obtained using: `http://lrde.epita.fr/~adl/git/spot.git/`.
A snapshot in which all our modifications are included can also be found at the following location: `http://lrde.epita.fr/dload/spot/spot-snapshot.tar.gz`

Chapter 1

Preliminaries

1.1 Linear Temporal Logics

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. In LTLs, properties like *eventually* or *never* are specified using special temporal operators.

In Spot, we use LTLs to describe sequences of transitions between states of the system, i.e. specifications about the system – as SPIN (Holzmann, 1990, 2003).

1.1.1 Definitions

A state of the system can be seen as the valuation of a set AP of atomic propositions. For example, the state of a traffic light can be modeled by the three propositions r (red), o (orange), and g (green) which tell whether the corresponding light is on: $\mathrm{AP} = \{r, o, g\}$. The possible executions of the system (state sequences) are infinite even though the systems we consider have a finite number of states, therefore we use infinite words to represent them.

Definition 1 *An infinite word over an alphabet Σ is a function $\sigma : [\![0, n[\![\to \Sigma$, where $n \in \mathbb{N} \cup \{\omega\}$ is an ordinal number.*

The ith letter of an infinite word σ is thus $\sigma(i)$.

Definition 2 *The suffix of an infinite word σ starting at the position i is the word denoted by σ^i such that $\forall j \in [\![0, n-i[\![, \sigma^i(j) = \sigma(i+j)$.*

We will only consider infinite words over $\Sigma = 2^{\mathrm{AP}}$, the power set of AP. An element of 2^{AP} represents a system configuration (a state) while an infinite word on 2^{AP} describes the systems evolution over the times.

Example 1 *Let's consider a traffic light model using three propositions to describe the state of the three lights. $\mathrm{AP} = \{r, o, g\}$, and 2^{AP} is thus the set $\Sigma = \{rog, ro\bar{g}, r\bar{o}g, r\bar{o}\bar{g}, \bar{r}og, \bar{r}o\bar{g}, \bar{r}\bar{o}g, \bar{r}\bar{o}\bar{g}\}$[1]. For example, the infinite word $\bar{r}o\bar{g} \cdot \bar{r}\bar{o}\bar{g} \cdot \bar{r}o\bar{g} \cdot \bar{r}\bar{o}\bar{g} \cdot \bar{r}o\bar{g} \cdot \bar{r}\bar{o}\bar{g} \cdots$ corresponds to a traffic light blinking orange, and matches the ω-regular expression $(\bar{r}o\bar{g} \cdot \bar{r}\bar{o}\bar{g})^{\omega}$.*

The meaning of a (E)LTL formula will always be determined with respect to an infinite word.

[1] Because an element of 2^{AP} can be seen as a list of propositions, for example here $\{r, o\}$ is equivalent to $ro\bar{g}$.

1.1.2 LTL

Syntax LTL formulæ are defined inductively from temporal operators X (next) and U (until) as follows:

- Every proposition $p \in \mathrm{AP}$ is a formula.
- If f_1 and f_2 are formulæ, then $\neg f_1$, $\mathsf{X}\, f_1$, $f_1 \wedge f_2$ and $f_1 \,\mathsf{U}\, f_2$ are formulæ.

Semantics Satisfaction of a LTL formula f w.r.t. the infinite word σ is denoted $\sigma \models f$.

- $\sigma \models p$ iff $p \in \sigma(0)$.
- $\sigma \models \neg f_1$ iff not $\sigma \models f_1$.
- $\sigma \models f_1 \wedge f_2$ iff $\sigma \models f_1$ and $\sigma \models f_2$.
- $\sigma \models \mathsf{X}\, f_1$ iff $\sigma^1 \models f_1$.
- $\sigma \models f_1 \,\mathsf{U}\, f_2$ iff $\exists i \geq 0$ such that $\sigma^i \models f_2$ and $\forall j \in [\![0, i-1]\!]$, $\sigma^j \models f_1$.

Usual temporal operators F (eventually), G (always), and R (release) are defined as the following abbreviations:

$$\mathsf{F}\, f_1 = \top \,\mathsf{U}\, f_1 \tag{1.1}$$

$$f_1 \,\mathsf{R}\, f_2 = \neg(\neg f_1 \,\mathsf{U}\, \neg f_2) \tag{1.2}$$

$$\mathsf{G}\, f_1 = \neg\,\mathsf{F}\,\neg f_1 = \neg(\top \,\mathsf{U}\, \neg f_1) = \bot \,\mathsf{R}\, f_1 \tag{1.3}$$

Example 2 *Here are some LTL formulæ based on the traffic light model.*

1. *The traffic light is not always* red.
   $\neg\,\mathsf{G}(r \wedge \neg o \wedge \neg g)$
2. *Every* orange *configuration is followed by a* red *one.*
   $\mathsf{G}((\neg r \wedge o \wedge \neg g) \rightarrow \mathsf{X}(r \wedge \neg o \wedge \neg g))$
3. *The system is infinitely often in the* green *configuration.*
   $\mathsf{G}\,\mathsf{F}(\neg r \wedge \neg o \wedge g)$

Because LTL and TGBA are not expressively equivalent, LTL might not be enough to specify all the properties we would like about our model like "the traffic light blinks orange" here.

1.1.3 ETL

Vardi and Wolper (1994) described different extensions of LTL where the temporal operators are defined by finite automata similarly to the extended LTL of Wolper et al. (1983).

Syntax The set of ELTL formulæ is defined inductively as follows:

- Every proposition $p \in \mathrm{AP}$ is a formula.
- If f_1 and f_2 are formulæ, then $\neg f_1$ and $f_1 \wedge f_2$ are formulæ.
- If $f_1, \ldots, f_n$ are formulæ, then $A(f_1, \ldots, f_n)$ is a formula, for every nondeterministic finite automaton (NFA) $A = (\Sigma, Q, I, F, \delta)$, where $\Sigma = \{\top, f_1, \ldots, f_n\}$, Q is the set a states, $I \subseteq Q$ is the set of initial sets, $F \subseteq Q$ is the set of accepting states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. We call $A(f_1, \ldots, f_n)$ an *automaton operator*.

Semantics By analogy with Section 1.1.2, we define the satisfaction of a ELTL formula.

- $\sigma \models p$ iff $p \in \sigma(0)$.
- $\sigma \models \neg f_1$ iff not $\sigma \models f_1$.
- $\sigma \models f_1 \wedge f_2$ iff $\sigma \models f_1$ and $\sigma \models f_2$.
- $\sigma \models A(f_1, \dots, f_n)$ iff $\exists$ an accepting run $r = s_0, s_1, \dots$ of $A(f_1, \dots, f_n)$ over σ.

Depending on how we define the accepting runs of $A(f_1, \dots, f_n)$, we get different versions of the logic. The different possible acceptances are: *repeating acceptance* when some state $s \in F$ occurs infinitely often in r, *finite acceptance* when some state $s \in F$ occurs in r, and *looping acceptance* when s is infinite. These three logics are independently as expressive as ω-regular expressions, and thus as Büchi automata (Vardi and Wolper, 1994), but they do not provide the same ease of expression of properties. Currently, Spot only recognizes ELTL with *finite acceptance* and *looping acceptance*, the difference being made depending on whether a given automaton operator has accepting states or not, and for now on we will only consider this ELTL.

Usual LTL temporal operators next, until, and globally can be defined by the ELTL automaton operators on Figure 1.1 below. G is *loop accepting* while U and X are *finite accepting*.

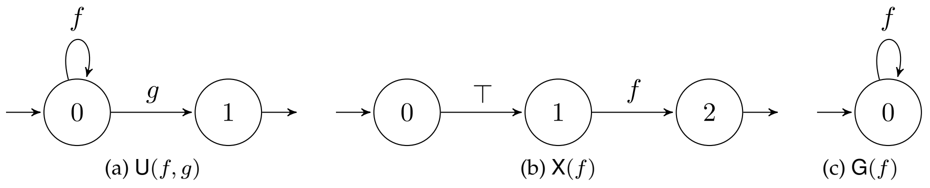

Figure 1.1: Automaton operators for LTL operators next, until and globally.

Example 3 *The formula* $f = \mathsf{G}\,\mathsf{F}(\neg r \wedge \neg o \wedge g)$ *based on the traffic light model can be represented in ELTL using automaton operators. Indeed,* $f = \mathsf{G}(\top\,\mathsf{U}(\neg r \wedge \neg o \wedge g))$ *in LTL (using Equation 1.1) which is perfectly valid in ELTL considering the automaton operators for* U *and* G *given above.*

Example 4 *The assertion that "the proposition* p *holds at least in every other state on a path" can be represented by the automaton operator* Blink *on Figure 1.2 and is thus expressible in ELTL.*

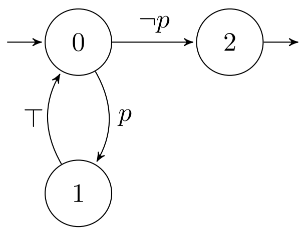

Figure 1.2: Automaton operator Blink(p) equivalent to $(p\top)^\omega$.

1.2 TGBA

Vardi (1986) introduced *the automata theoretic approach to model checking of LTL formulæ* based on Büchi automata – a special kind of ω-automaton. A Büchi automaton accepts an infinite input sequence iff there exists a run of the automaton which visits (at least) one of the final states infinitely often.

1.2.1 Definition

Spot uses TGBA, which are Büchi-automata with labels on transitions, and multiple acceptance sets of transitions. Compared to Büchi Automata with acceptance sets of states, TGBA allow LTL formulæ to be represented more concisely.

Definition 3 *A TGBA is a quintuplet $A = (\Sigma, Q, I, F, \delta)$, where Σ is an alphabet, Q is the set a states, $I \subseteq Q$ is the set of initial states, F is a finite set of acceptance conditions, and $\delta \subseteq Q \times (2^{\Sigma} \setminus \{\varnothing\}) \times 2^{F} \times Q$ is the transition relation. (Each transition is labeled by a set of acceptance conditions.)*

The acronym TGBA (Transition-based Generalized Büchi Automaton) was first introduced by Giannakopoulou and Lerda (2002). A TGBA accepts a run if it passes through (at least) one transition of every set of accepting transitions infinitely often.

We use elements of $2^{\Sigma} = 2^{2^{\mathrm{AP}}}$ to represent labels which is legitimate because there is a bijection between this set and the set of formulæ on AP. Indeed, $\{\{a, b\}, \{a\}\}$ can be seen as $(a \wedge b) \vee (a \wedge \neg b)$ or also simply as a which may seem more natural for an automaton.

Example 5 *The TGBA $\langle 2^{\{a,b\}}, \{q\}, \{q\}, \{f, g\}, \delta\rangle$, where*

$$\delta = \{(q, 2^{\{a,b\}}, \emptyset, q), (q, \{\{a, b\}, \{a\}\}, f, q), (q, \{\{a, b\}, \{b\}\}, g, q)\}$$

accepts the same language that the LTL formula $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$.

The two acceptance conditions f and g, represented on Figure 1.3 by "●" and "●", ensure that every path passes infinitely often accepted by both of these transitions.

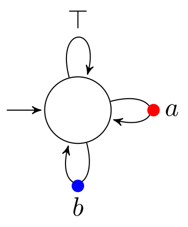

Figure 1.3: TGBA corresponding to $\mathsf{G}\,\mathsf{F}\,a \wedge \mathsf{G}\,\mathsf{F}\,b$.

1.2.2 Symbolic representation

This section explains how a TGBA can be represented in a symbolic way—instead of the usual explicit representation—using *binary decision diagrams* (BDDs) (Andersen, 1997; Bryant, 1992). That is interesting because the LaCIM algorithm presented in Chapter 2 constructs directly a

TGBA in a symbolic way from a (E)LTL formula. There is some advantages to have a symbolical reprensentation (among others): first, it is less memory-consuming (because of BDDs) and second, some algorithms can be directly applied on the Boolean functions reprensenting the TGBA instead of doing a traversal in the explicit case.

A BDD is a data structure providing a canonical and efficient way to represent and manipulate Boolean formulæ, i.e. $a \to (b \wedge c)$ and $(\neg a) \vee (b \wedge c)$ have the same BDD representation. They have been used in numerous symbolic model checking algorithms (Burch et al., 1990).

Symbolically the transition relation δ and the sets of states Q and I of a TGBA are seen as Boolean formulæ (and efficiently represented as BDDs) by identifying states and acceptance conditions by vectors of Boolean variables. We use as many Boolean variables as necessary to have each state unique, and for each Boolean variable b we define another variable named b' which is used to represent destination states in transitions. We also use as many Boolean variables as necessary to have each acceptance condition unique.

Example 6 *The TGBA on Figure 1.4 has three states and one acceptance condition, so we need two Boolean variables for the states (b_1 and b_2), and one for the acceptance condition (a_1).*

- *The set of states is* $Q = b_1b_2 \vee \bar{b}_1b_2 \vee b_1\bar{b}_2$,
- *The set of initial states is* $I = b_1b_2$,
- *The transition relation δ is represented by two Boolean formulæ: T for the transitions and Acc for the acceptance conditions which are as follows:*
  $T = b_1b_2p_1b'_1\bar{b}'_2 \vee b_1\bar{b}_2\bar{p}_1\bar{b}'_1b'_2 \vee \bar{b}_1b_2p_2b'_1b'_2$,
  $Acc = b_1b_2p_1a_1$.

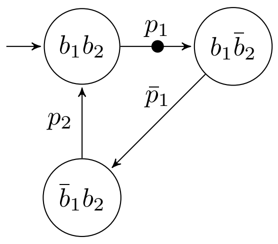

Figure 1.4: A TGBA where states are vectors of Booleans.

This symbolic representation completely defines a TGBA and can be used to do whatever we want on it using BDD operations. For example on the example above, we can do the following:

- $T \wedge I = b_1b_2p_1b'_1\bar{b}'_2$ gives the outgoing transitions of the initial state,
- $(\exists b_1 \exists b_2 \exists p_1 (b_1b_2p_1b'_1\bar{b}'_2))[b_1/b'_1][b_2/b'_2] = b_1\bar{b}_2$ gives the destination state of the previous transition, where $f[x/y]$ denotes the Boolean formula obtained by substituting every free occurrence of y by x in the Boolean formula f,
- $\exists b_1 \exists b_2 \exists p_1 ((b_1b_2p_1b'_1\bar{b}'_2) \wedge Acc = a_1)$ gives its acceptance conditions.

Chapter 2

LaCIM Algorithm

Many articles deal with the construction of ω-automata from a LTL formula, and very good states of the art can be found in Duret-Lutz (2007) and Wolper (2000).

In Spot, two translations of LTL formulæ into TGBA are implemented: FM (Couvreur, 1999) and LaCIM (Couvreur, 2000). This algorithm also having an ELTL version we decided to implement it in Spot, and the next two sections present that algorithm for both logics.

2.1 LTL

This version uses a set $el(f)$ which characterises *elementary formulæ* of the LTL formula f, i.e. subformulæ of f (including f itself) involved in the translation, and is defined as follows:

$el(f) = \{f\} \cup \{g \,\mathsf{U}\, h \mid g \,\mathsf{U}\, h \in sub(f)\} \cup \{g \mid \mathsf{X}\, g \in sub(f)\}$, where $sub(f)$ denotes the set of subformulæ of f which is trivially defined inductively.

LaCIM is based the following assertion: it is possible to construct a TGBA representing f by using states labeled by conjunction of elementary formulæ. More precisely, this TGBA can be (naively) constructed from a LTL formula as follows:

1. Represent all possible states, i.e. those labeled by a conjunction of compatible elementary subformulæ.
2. Represent all compatible transitions between states, i.e. states labeled by $\mathsf{X}\, g$ can only have successors labeled by g, and states labeled by an $g \,\mathsf{U}\, h$ can have only successors labeled by $g \,\mathsf{U}\, h$ (if g is true now) or labeled by anything (if h is true now).
3. Define initial states, i.e. those labeled by (at least) the input formula.
4. Define accepting transitions, i.e. those respecting the premises made by until operators.

A full example for the formula $f = \mathsf{X}(p \,\mathsf{U}\, q)$, with the four steps detailed, is available next page on Figure 2.1, where $p \,\mathsf{U}\, q$ is denoted g in order to simplify.

This algorithm is well suited for symbolical representations, and the following formally present it as directly constructing a symbolical TGBA.

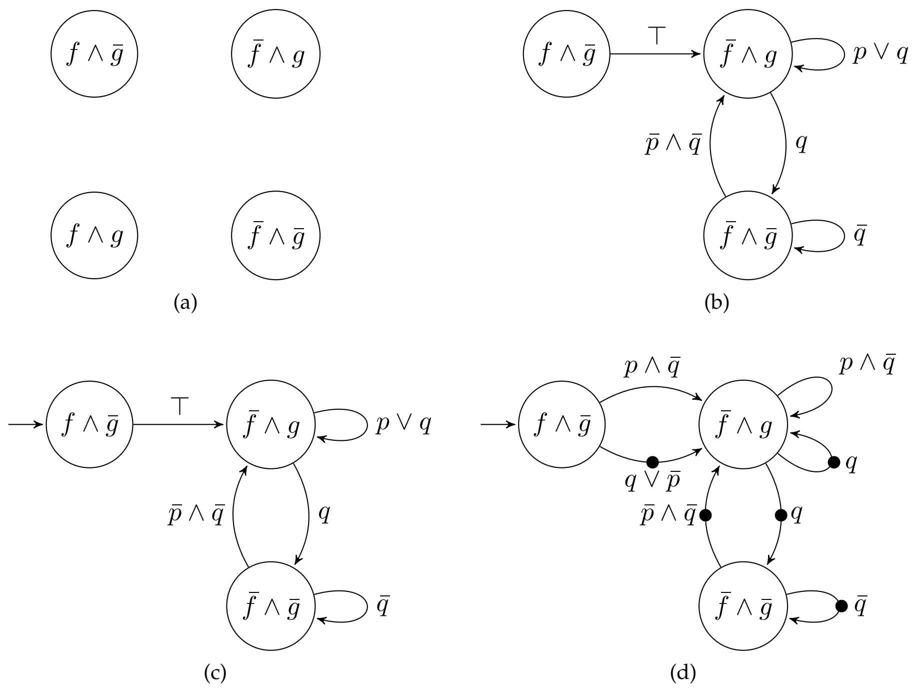

Figure 2.1: Construction of a TGBA for $f = \mathsf{X}(p \,\mathsf{U}\, q)$.

The symbolical algorithm uses two Boolean variables for each elementary formula g: one denoted $Now[g]$ (which means that g is true at the current moment), and one denoted $Next[g]$ (which means that g is true at the next moment). It also uses one Boolean variable per acceptance condition (i.e. per subformula like $g \,\mathsf{U}\, h$) denoted $Acc[g \,\mathsf{U}\, h]$; those variables are then used to construct the Boolean function Acc (this part is not explained here to simplify). Only Now variables are used to represent the different states.

The resulting constructed TGBA from an LTL formula f is defined as follows:

- $I = Now[f]$,
- $T = \bigwedge\limits_{g \,\mathsf{U}\, h \in el(f)} (Now[g \,\mathsf{U}\, h] \Leftrightarrow (\phi(h) \vee (\phi(g) \wedge Next[g \,\mathsf{U}\, h]))) \wedge \bigwedge\limits_{\mathsf{X}\, g \in el(f)} Now[g] \Leftrightarrow \phi(g)$,
- $\forall g \,\mathsf{U}\, h \in el(f), Acc[g \,\mathsf{U}\, h] = \neg Now[g \,\mathsf{U}\, h] \vee \phi(h)$.

The Boolean function ϕ being defined inductively as follows:

- $\phi(p) = p$, for every proposition $p \in AP$.
- $\phi(\neg f_1) = \neg\phi(f_1)$
- $\phi(f_1 \wedge f_2) = \phi(f_1) \wedge \phi(f_2)$
- $\phi(\mathsf{X}\, f_1) = Next[f_1]$
- $\phi(f_1 \,\mathsf{U}\, f_2) = Now[f_1 \,\mathsf{U}\, f_2]$

The Boolean formulæ T constrains the possible transitions in order to respect the semantics of X and U in LTL (step 2), while Acc constrains the acceptance conditions of the resulting TGBA in order to respect the fact that for every subformula $g \,\mathsf{U}\, h$, h has to be true at a present or future moment (step 4).

Example 7 *Here is the TGBA corresponding to the formula $f = p \,\mathsf{U}\, q$.*

- $I = Now[f]$
- $T = Now[f] \Leftrightarrow q \vee (p \wedge Next[f])$
- $Acc[f] = \neg Now[f] \vee q$

Figure 2.2 gives a graphic reprensentation of the automaton. But we do not need this explicit representation to perform operations on that automaton (as we have seen in Section 1.2.2).

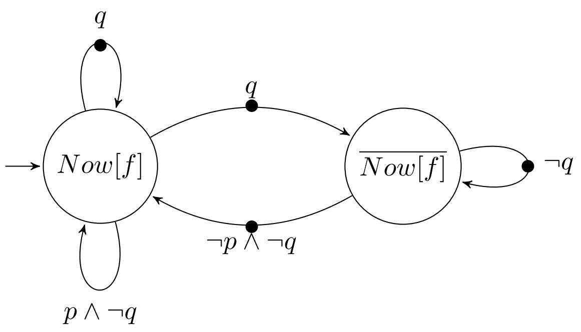

Figure 2.2: TGBA corresponding to $f = p \,\mathsf{U}\, q$.

2.2 ELTL

This version is based on the same assertion as the previous one but elementary formulæ are defined as follows:

$el(f) = \{A_s \mid A \in automatop(f), s \in Q\}$, where $automatop(f)$ denotes the set of automaton operators appearing in f and Q denotes the set of states of A.

The naive construction can be modified as follows:

2. Represent all compatible transitions between states, i.e. states labeled by A_s can only have successors labeled by A_r (if r is a successor of s in A labeled by a formula f which is true now) or labeled by anything (if there exists a final state r successor of s in A labeled by a formula f which is true now).

4. Define accepting transitions, i.e those respecting premises made by automaton operators (an accepting run has to be found).

The symbolic algorithm uses four Booleans variables for each elementary formula A_s: one denoted $Now[A_s]$ (which means that the automaton operator A is currently in the state s), and one denoted $Next[A_s]$ (which means that the automaton operator A is in the state s at the next moment), plus two denoted $Now[A_s^\alpha]$ and $Next[A_s^\alpha]$. It also uses one Boolean per acceptance condition (i.e. per automaton operator A) denoted $Acc[A]$. As in the LTL version, only Now variables are used to represent the different states.

As we have seen in Section 1.1.3, there is (in the ELTL we consider) two possible acceptances for automaton operators: *finite* and *looping*, and we have to ensure that accepting conditions are well defined in both cases, i.e. to make sure that an accepting run is found. In the *finite* case, when an accepting run is found, it is valid to be accepting afterwards because we are not trying to find one anymore. In order to do that, we use the alpha variables defined above: we use $Now[A_s^\alpha]$ as meaning "we are not verifying A_s" which is always false but in states which have a predecessor labeled by A_r where r is a final state. The *looping* case being defined dually.

The algorithm is thus slightly different depending on the acceptance: Acc variables are not contrained in the same way, and the operator $\odot$ used below is actually $\Rightarrow$ in the *finite* case and $\Leftarrow$ in the *looping* case.

The resulting constructed TGBA from an LTL formula f is defined as follows:

- $I = Now[f]$,

- $$\begin{aligned} T = &\bigwedge_{A_s}(Now[A_s] \Leftrightarrow \bigvee_{s \xrightarrow{g} r, r\notin F}(\phi[g] \wedge Next[A_r]) \vee \bigvee_{s \xrightarrow{g} r, r\in F} \phi[g])\wedge \\ &\bigwedge_{A_s}(Now[A_s^\alpha] \odot \bigvee_{s \xrightarrow{g} r, r\notin F}(\phi[g] \wedge Next[A_r^\alpha]) \vee \bigvee_{s \xrightarrow{g} r, r\in F} \phi[g])\wedge \\ &\bigwedge_{A_s}(Acc[A] \Rightarrow \bigwedge_{s\in Q}(Next[A_s^\alpha] \Leftrightarrow Next[A_s]), \end{aligned}$$

  For *finite* automaton operators:

- $\forall A, Acc[A] = \bigwedge_{s\in F}(\neg Now[A_s^\alpha] \vee \bigvee_{s \xrightarrow{g} r, r\in F} \phi[g]),$

  For *looping* automaton operators:

- $\forall A, Acc[A] = \bigwedge_{s\in F}(Now[A_s^\alpha] \vee \neg \bigvee_{s \xrightarrow{g} r, s\in Q} \phi[g]).$

The Boolean function ϕ being defined inductively as follows:

- $\phi(p) = p$, for every proposition $p \in AP$.
- $\phi(\neg f_1) = \neg\phi(f_1)$
- $\phi(f_1 \wedge f_2) = \phi(f_1) \wedge \phi(f_2)$
- $\phi(\mathsf{A}) = Now[A_i]$, where i is the initial state of A. There is an abuse here (and before) because we implicitly consider the arguments as part of the automaton operators but it actually simplifies the notations.

Chapter 3

Implementation

The first part of the implementation was to integrate ELTL formulæ in Spot, which means first write AST (Abstract Syntax Tree) classes to represent them in memory, and secondly a parser that instanciates an AST from a formula given in a file or by a string, as it was already the case for LTL formulæ. Then we could implement LaCIM for ELTL to make our extended LTL actually usable in Spot. This is presented in the following sections. Spot is a C++ library, and thus everything here might be considered from a C++ point of view.

3.1 AST

LTL and ELTL formulæ only differ in the way temporal operators are defined, and thus we wanted to maximize sharing between LTL and ELTL ASTs.

A first idea was to define two specific ASTs which cannot be instanciated in any way to represent invalid formulæ. But in that case, it is not straightforward to share classes which are used in both logics, for example the class representing binary operators in LTL cannot be used directly because U is not *per se* a valid binary operator in ELTL (it's an automaton operator). In order to solve this, we wrote *generic* versions of existing LTL nodes that are also present in ELTL using templates, which was a lot of rewriting. Furthermore, this brought us new issues: first, some parts of the library were broken by the use of templates like the python wrapper which uses SWIG. Second, considering that ELTL ASTs have one more node (for automaton operators), we were not able to share visitors between both logics directly either. We finally decided to drop this idea and to revert all modifications.

Instead, we decided to integrate our new node in the existing LTL AST hierarchy, which is now able to reprensent both logics. We just had to handle the cases in visitors and algorithms where a given node is not possible for either of the logics. Only few modifications where made without breaking anything.

3.2 Parser

The parser reads formulæ from a file in a specific format, and is implemented using Bison in the `src/eltlparse` directory of Spot. A few tests were also written in `src/eltltest`.

Define an ELTL formula means two things: first, define all automaton operators we want to use and second, define the actual ELTL formula using those automaton operators. Automaton operators are basically NFAs which alphabet is the set of all arguments of that operator plus the constant *true* (cf. Section 1.1.3), which means that we need to abstract the arguments in the definition of automaton operators.

A ELTL file is split in two parts delimited by the '%' character: the definition of the automaton operators and the ELTL formula. The Figure 3.1 below shows a simple file for the constant *true* without any automaton operators defined. Comments are available using '#'.

```
# Automaton operators definition.
%
# ELTL Formula.
1
```

Figure 3.1: Simple file for the constant *true*.

Automaton operators Syntactically, an empty meaningless automaton operator is simply defined by '`A=()`', and used in the formula with '`A()`'. Transitions composed of a source state, a destination state and a label need to be added to define actual automaton operators. States are represented as integers and labels are either arguments of the automaton operator (denoted by $\$0, \$1, \ldots$) or the constant *true*. Thus, a transition from the state 0 to the state 1 labeled by the first argument is defined by '`0 1 $0`'. A state s can be defined as accepting by adding the line '`accept s`', in this case the operator is no longer *finite accepting* but *loop accepting* as discussed in Section 1.1.3. A complete example is represented on Figure 3.2 below.

```
A=(
    0 1 $0
    1 2 true
    0 0 $1
    accept 2
  )
%
A(f,g)
```

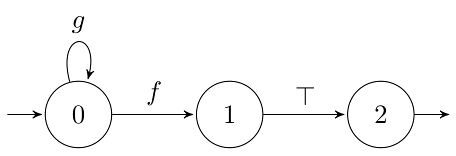
$$\begin{tikzcd} \arrow[r] & 0 \arrow[loop above, "g"] \arrow[r, "f"] & 1 \arrow[r, "\top"] & 2 \arrow[r] & {} \end{tikzcd}$$

Figure 3.2: ELTL formula file, with '`A(f,g)`' representation.

The source state of the first transition becomes the initial state, and there is no default accepting state. All binary operators can also be used in infix notation like U in Figure 3.4.

Aliases As it was already said, some temporal operators are defined as abbreviations of other ones like $\mathsf{F}\, a = \top \,\mathsf{U}\, a$ (cf. Equation 1.1). It seems natural to be able to define them this way as well in ELTL files. Thus we implemented the possibilty to define aliases of automaton operators as shown on Figure 3.3.

ELTL formulæ All usual logical connectives can be used in a ELTL formula, and this part of the parser is greatly inspired from the LTL parser of Spot. The only major difference is that unary operators cannot be used without parenthesis around their argument, which would be ambiguous for the ELTL parser considering the fact that operators' names are unknown.

Include In order to avoid having to redefine common temporal operators, as those present in LTL, the parser provides an include directive. The ltl_defs file Figure 3.3 defines usual LTL operators (cf. Figure 1.1) while Figure 3.4 uses some of them.

```
# Finite acceptance.
X=(
    0 1 true
    1 2 $0
    accept 2
  )
U=(
    0 0 $0
    0 1 $1
    accept 1
  )

# Looping acceptance.
G=(
    0 0 $0
  )

# Aliases.
F=U(true, $0)
R=!U(!$0, !$1)
Weak=G(F($0))
Strong=G(F($0))->G(F($1))
```

Figure 3.3: Usual LTL temporal operators file: ltl_defs.

```
include ltl_defs
%
X(f) & f U g | Strong(f,h)
```

Figure 3.4: ELTL file using ltl_defs.

We could also have defined G as an alias ($\mathsf{G} = \mathsf{R}(\textbf{false}, \$0)$) which would have given us exactly the same translated TGBA for any ELTL formula using G.

3.3 LaCIM

LaCIM (ELTL) is implemented – like its LTL equivalent – as a const visitor but on ELTL formulæ and is avalaible in `src/tgbaalgos/eltl2tgba_lacim.hh`.

Correctness LaCIM is a rather short but a very complex algorithm and to be sure that we implemented it correctly, we included it with other translation algorithms in tests using LBTT in Spot. Indeed, LBTT (Tauriainen and Heljanko, 2002) is a tool to check and compare translation algorithms from LTL to Büchi automata. It works by comparing results of each algorithm with each other on random formulæ, and by using them to *model check* randomly generated states spaces. It is very useful to find errors, and we managed to make our implementation correct with it after a lot of rewriting.

Benchmarks The Table 3.1 below presents benchmarks between the different translation algorithms available in Spot. LBTT was used to make those measurements on random formulæ which were obtained by using the script `src/tgbatest/spotlbtt.test` of Spot.

	Automaton		Product		
Algorithm	states	transitions	states	transitions	Times
FM	657	1207	11377	48805	5.85
LaCIM (LTL)	1304	6364	25172	154034	6.24
LaCIM (ELTL)	24917	80642	336166	2317071	12.46

Table 3.1: Benchmarks between translation algorithms in Spot

Optimizations Clearly LaCIM for ELTL is far behind other algorithms. We have some ideas to improve it which are not yet implemented by lack of time; it is some future work to do.

Chapter 4

ELTL & PSL

LTL is not as expressive as we would like it to be; that was our first motivation to implement ELTL in Spot. But LTL was extended in a different fashion in PSL, and this chapter investigates the similarities between those extensions.

PSL (Property Specification Language) is a property specification language developed by Accellera and recently standardized by the IEEE. It includes as its temporal layer a linear temporal logic that enhances LTL with regular expressions and other useful features (PSL, 2004). This temporal layer uses so-called SEREs (Sequential Extended Regular Expressions) to define sequences (such as repetitions) built from Boolean expressions. Considering that ELTL temporal operators are based on NFAs, we thought it might be possible to integrate in it some SERE operators, something that is not possible in LTL where sequences cannot be actually defined. The following sections present some interesting SERE operators and how we can integrate them in our ELTL.

4.1 SERE operators

In LTL vocabulary, we can see SEREs operators as temporal operators which are defined from atomic propositions and formulæ built from them like usual LTL operators. PSL is a complex language and saying that might be considered as an abuse but from our LTL view point we are only interested here by SEREs as temporal operators. That is why we will consider infinite paths here as in LTL, and *hold on a path* shall thus be understood as *hold on a path when it makes sense*. We will see how we could easily handle that problem in our integration in the next section.

Concatenation (;) The expression: `a;b` holds on a path iff there is a future position i such that a holds on the path up to and including the position i and b holds on the path starting at the position $i + 1$. For example, the expression `a`U`b;`X`c` holds on `aaabac...`

When a and b are simple atomic propositions, `a;b` is equivalent to the LTL formula $a \wedge \mathsf{X}\ b$. But when a does not hold on only one position, it is not possible anymore to define the concatenation in LTL because we would need to know the position i to write the proper number of X before b.

Fusion (:) The expression: `a:b` holds on a path iff there is a future position i such that a holds on the path up to and including the i^{th} position and b holds on the path starting at the i^{th} position. One can note that `a:`X`b` is the same as `a;b`.

Length-matching and (&&) The expression: `a && b` holds on a path iff both a and b hold at the current position, and furthermore both complete at the same position.

Within The expression: `a within b` holds iff b holds at the current position, and a starts at or after the position in which b starts, and completes at or before the position in which b completes.

Consecutive repetition ([*]) This operator constructs repeated consecutive concatenation. For example `a;a;a` is equivalent to `a[*3]`. The number of repetition can also be a range as in: `a[*n:m]` which holds on a path iff the path can be partitionned into between n and m parts, inclusive, where a holds on each part. It is also possible not to specify any number: `a[*]`.

More operators do exist but are either already present in LTL or not adaptable at our context (like those using PSL clocks).

4.2 Integration in ELTL

Intuitively, if we want to express SERE operators in our ELTL, we need to be able to know if a given formula just completed; something which is not possible so far in our format. Let $\mathcal{F}(a)$ be the formula saying that a just finished (i.e. completed), which can be defined as follows:

$$\mathcal{F}(a) = \begin{cases} \displaystyle\bigvee_{s \xrightarrow{g} r, r \in F} (Now[A_s] \wedge \phi[g]), & \text{if } a \text{ is an automaton operator} \\ \top, & \text{otherwise} \end{cases}$$

With that formula, SERE operators can be defined as the following automaton operators:

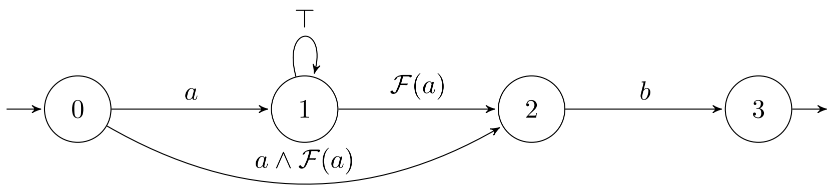

Figure 4.1: `a;b`

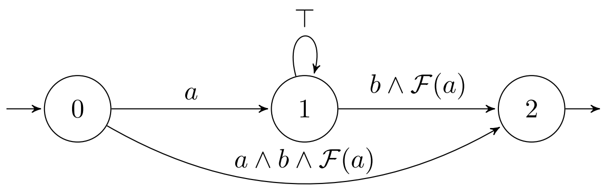

Figure 4.2: `a:b`

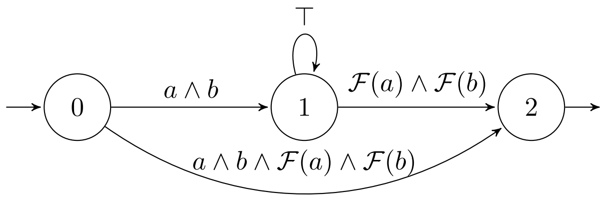
⊤
$a \wedge b$
$\mathcal{F}(a) \wedge \mathcal{F}(b)$
$a \wedge b \wedge \mathcal{F}(a) \wedge \mathcal{F}(b)$
0
1
2

Figure 4.3: `a && b`

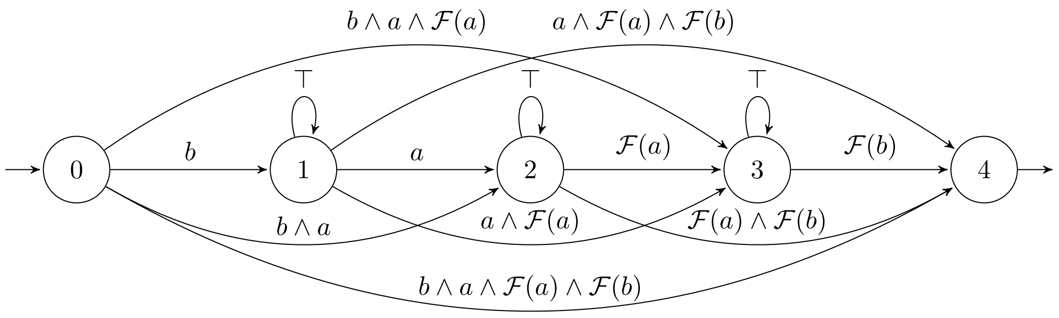
$b \wedge a \wedge \mathcal{F}(a)$
$a \wedge \mathcal{F}(a) \wedge \mathcal{F}(b)$
⊤
⊤
⊤
b
a
$\mathcal{F}(a)$
$\mathcal{F}(b)$
0
1
2
3
4
$b \wedge a$
$a \wedge \mathcal{F}(a)$
$\mathcal{F}(a) \wedge \mathcal{F}(b)$
$b \wedge a \wedge \mathcal{F}(a) \wedge \mathcal{F}(b)$

Figure 4.4: `a within b`

In order to be able to express those automaton operators, a *finish* keyword (representing $\mathcal{F}$) was added to the ELTL parser. The Figure 4.5 below shows an example using it.

```
W=(
    0 1 $0
    1 2 finish($0)
    accept 2
  )
```

Figure 4.5: A temporal operator wrapping any other one using *finish*.

Because conjunctions of arguments are not possible directly in the definiton of an automaton operator – but only in aliases – it is not straightforward to define PSL operators in ELTL. There is thus some future work to do to actually integrate them in our ELTL in Spot.

Also, even if the consecutive repetition operator can be easily defined as an automaton operator when n and m are known (`a[*n:m]`) or when no number is specified (`a[*]`), this operator does not fit very well in our ELTL format because of those integers n and m. Considering that a *generic* version doesn't seem possible, another future work is thus to investigate how we could integrate it anyway.

Conclusion

The work we presented allows Spot to verify properties express in an extended temporal linear logic (ELTL). In order to do that, we first wrote AST (Abstract Syntax Tree) classes to represent this new logic in memory. Then, we implemented a parser that reads an ELTL formula given in a file or by a string. Finally, the LaCIM translation algorithm to TGBA for this extended logic was written. Everything which has been done is based on the existing LTL implementation, making the library consistent. As mentioned before, there is still some work to do on LaCIM for ELTL to reduce the size of the TGBA produced.

A simple test program which uses everything we have implemented was also written in the following file: `src/tgbaalgos/eltl2tgba.cc`. It reads an ELTL formula from a string or a file, and outputs the TGBA corresponding to the formula constructed by LaCIM in the DOT language.

Similarities between ELTL and the temporal layer of PSL which enhances LTL with regular expressions were also investigated. We showed how some PSL operators using regular expressions could be defined in ELTL with automaton operators. However, this definition in ELTL is not easy to write because PSL operators do not fit very well in our ELTL format, and another future work is thus to investigate how we could integrate it more nicely.

One can use what has been done to check specifications of finite-state systems that were not expressible before with Spot.

Acknowledgements

I wish to thank my advisor Alexandre Duret-Lutz for valuable feedback and for his everyday time and patience to teach me.

Bibliography

Andersen, H. R. (1997). An introduction to binary decision diagrams. Lecture notes.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. (1990). Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C. IEEE Computer Society Press.

Couvreur, J.-M. (1999). On-the-fly verification of temporal logic. In Wing, J. M., Woodcock, J., and Davies, J., editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France. Springer-Verlag.

Couvreur, J.-M. (2000). Un point de vue symbolique sur la logique temporelle linéaire. In Leroux, P., editor, *Actes du Colloque LaCIM 2000*, volume 27 of *Publications du LaCIM*, pages 131–140, Montréal. Université du Québec à Montréal.

Duret-Lutz, A. (2007). *Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents*. PhD thesis, Université Pierre et Marie Curie.

Duret-Lutz, A. and Poitrenaud, D. (2004). Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83, Volendam, The Netherlands. IEEE Computer Society Press.

Giannakopoulou, D. and Lerda, F. (2002). From states to transitions: Improving translation of LTL formulæ to Büchi automata. In Peled, D. and Vardi, M., editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, Texas. Springer-Verlag.

Holzmann, G. J. (1990). *Design And Validation Of Computer Protocols*. Prentice Hall PTR.

Holzmann, G. J. (2003). *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

PSL (2004). Property specification language, reference manual.

Tauriainen, H. and Heljanko, K. (2002). Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70.

Vardi, M. Y. (1986). An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press.

Vardi, M. Y. and Wolper, P. (1994). Reasoning about infinite computations. *Information and Computation*, 115(1):1–37.

Wolper, P. (2000). Constructing automata from temporal logic formulas: A tutorial. In Brinksma, E., Hermanns, H., and Katoen, J.-P., editors, *Proceedings of the FMPA 2000 summer school*, volume 2090 of *Lecture Notes in Computer Science*, pages 261–277, Nijmegen, the Netherlands. Springer-Verlag.

Wolper, P., Vardi, M. Y., and Sistla, A. P. (1983). Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pages 185–194. IEEE Computer Society Press. Later extended and published as Vardi and Wolper (1994).