International Journal of Science and Engineering Applications
Volume 7–Issue 04, 54-63, 2018, ISSN:-2319–7560

EFFICIENT ALGORITHM TO TRANSFORM MINIMALIST SUBSET OF LTL FORMULA INTO FINITE STATE MODELS

Bilal Kanso
Lebanese University
Department of Computer Science, Beirut

Ali Kansou
Lebanese University
Department of Computer Science, Beirut

Abstract: The translation of LTL formula into equivalent *Büchi* automata plays an important role in many algorithms for LTL model checking, which consist in obtaining a *Büchi* automaton that is equivalent to the software system specification and another one that is equivalent to the negation of the property. The intersection of the two *Büchi* automata is empty if the model satisfies the property.

Generating 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. There is no polynomial solution for checking emptiness of the intersection. That comes from the translation step of LTL formula into finite state models. This makes verification methods hard or even impossible to be implemented in practice. In this paper, we propose a subset of LTL formula which can be converted to *Büchi* automata whose the size is polynomial.

Keywords: Linear Temporal Logic, *Büchi* automata, Model checking, Compositional modelling

1. INTRODUCTION

Model checking becomes increasingly one of the most important tools to verify the correctness of computer-based control systems [1, 4, 12, 15]. It is a formal verification technique consisting in algorithmically verifying whether system properties such as the absence of deadlocks (described in some appropriate logical formalism such as temporal logic) are satisfied by the system (described as a suitable finite state model). The success of the model checking technique comes from the fact that it is completely automatic. Running a model checking on a given system model to verify a desired property leads automatically to fail state or successful state. In case the system model fails to satisfy the property, the model checking tool can offer a counterexample which can be used as an error trace provided for debugging purposes.

Model checking approaches vary according to the logic used to specify system properties [3, 12, 18]. One of the most used logics is the Linear Temporal Logic (LTL) [11]. The underlying idea consists in transforming the negation of the LTL expression into a Büchi automaton, and then computing the product between the Büchi automaton representing the system and the one representing the negation of the LTL expression. If the product is not empty, that means the property expressed by the negation of the LTL expression is not satisfied by the system, otherwise the property is well-satisfied. However, the decision problem for emptiness of the intersection is PSPACE-hard [2, 19]. That comes from the translation of LTL formula into Büchi automata. Indeed, the space complexity of this approach is linear in the size of Büchi automata and exponential in the length of the LTL formula: the Büchi automaton of a property (described as a LTL formula) is constructed in exponential space in the length of this property. This makes verification methods hard or even impossible to be implemented in practice and makes the scalability of the LTL model checking limited, which commonly referred to as the state explosion problem [8].

In this paper, we contribute to finding a subset of LTL properties that can be converted polynomially into Büchi automata. Finding such a subset of LTL logic will be viewed as one the most promising directions to bridge the gap between the increasing complexity of state models and actual model checking methods. We define a fragment that we call, *Flat LTL Logic* and we show how formula in this fragment can be transformed into Büchi automata whose the state space size is linear. Due to the structure of flat LTL formula, our algorithm can be compositional in the sense that the final finite state model associated to a given formula is obtained by developing a sub-automaton for each sub-formula of the principal formula. Hence, the basic idea for developing the final automaton for a flat LTL formula f is that f can be recursively decomposed into a set of sub-formula, arriving at sub-formula that can be completely handled. Composition is then used for assembling different sub-automata and then forming larger ones. Such a composition can be seen as an operation taking sub-automata for sub-formula as well as the flat LTL operator to provide a new more complex automaton.

In order to guide the construction of the final automaton for a flat LTL formula f from the sub-automata associated to the sub-formula f_1, f_2, … , f_n of f, we build the finite syntax tree, *FST*(f) of the formula f. The nodes of a finite syntax tree are labeled, either by flat LTL operators or by propositional operators. The leaves are labeled only by atomic propositions.

Thus, the target Büchi automaton is obtained by exploring the tree in pre-order.

The rest of this article is organized as follows: Section 2 briefly describes Büchi automata. In Section 3, we describe our fragment of LTL logic and the reasons to choose it. In Section 4, we present for each formula in our fragment LTL, its equivalent Büchi automata and show the proof of this equivalence. Section 5 presents the finite syntax tree associated to a formula defined in our fragment LTL while Section 6 shows the final algorithm that generates to any formula in our fragment an equivalent Büchi automaton. Section 7 presents the conclusion and some future works.

2. Automata on infinite words

2.1 Büchi automata

Automata on infinite inputs were introduced by Büchi. A Büchi automaton is a non- deterministic finite-state automaton which takes infinite words as input [9, 10, 14]. A word is accepted if the automaton goes through some designated "good" states infinitely often while reading it. A **Büchi automaton** is a finite state automaton defined by a 5-tuple $A = (S, s_0, F, \sum, \delta)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- Σ is a non-empty set of atomic propositions,
- $F \subseteq S$ is a finite set of accepting states,
- $\Delta : S \times \Sigma \to 2^S$ is a transition function.

In the following of this paper, the initial state of a Büchi automaton is pointed to by incoming arrows and the accepting states are marked by double circles.

A run of A on $\sigma = \sigma(0)\sigma(1)\sigma(2) \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0s_1s_2 \ldots \in S^\omega$ starting with the initial state s_0 of A such that $\forall i, i \geq 0$, $s_{i+1} \in \delta(s_i, \sigma(i))$. A run $s_0s_1s_2 \ldots$ is **accepting** by an automaton A if A goes through accepting states (*i.e.* $\in F$) infinitely often while reading it. The *accepted language* of a Büchi automaton A, denoted by $\mathscr{L}_\omega(A)$, is then defined by:

$$\mathscr{L}_\omega(A) = \{ \sigma \text{ in } \Sigma^\omega \mid \text{there is an accepting run for } \sigma \text{ in } A \}$$

2.2 Operations on Büchi automata

The basic idea of the construction of the union of two Büchi automata A_1 and A_2 is to add a new initial (nonaccept) state s_{new} to the set of states union of A_1 and A_2. The transitions of the union of A_1 and A_2 are transitions of both A_1 and A_2 with the following two transitions:

a) A transition from s_{new} to a state s labeled with a proposition p if and only if there is transition from the initial state of A_1 to the state s labeled with the proposition p;

b) A transition from s_{new} to a state s labeled with a proposition p if and only if there is transition from the initial state of A_2 to the state s labeled with the proposition p

Definition 1 (Buchi automata union). *Let $A_1 = (S_1, s_{10}, F_1, \Sigma, \delta_1)$ and $A_2 = (S_2, s_{20}, F_2, \Sigma, \delta_2)$ be two büchi automata. The union $A_1 \cup A_2$ of A_1 and A_2 is the büchi automaton $A = (S, s_0, F, \Sigma, \delta)$ defined as follows:*

- $S = S_1 \cup S_2 \cup \{s_0\}$
- *$s_0 \in S$ is the initial state*
- $F = F_1 \cup F_2$
- *the transition relation δ is defined as follows:*

$$\delta(s,p) = \begin{cases} \delta_1(s,p) \text{ if } s \in S_1 \\ \delta_2(s,p) \text{ if } s \in S_2 \\ \delta_1(s_{10},p) \cup \delta_2(s_{20},p) \text{ if } s \text{ is the initial state } s_0 \end{cases}$$

The construction of the intersection automaton works a little differently from the finite state automata case. One needs to check whether both sets of accepting states are visited infinitely often. Consider two runs r_1 and r_2 and a word σ where r_1 goes through an accept state after $\sigma(0)$, $\sigma(2)$, $\cdots$ and r_2 enters accept state after $\sigma(0)$ $\sigma(3)$ $\cdots$. Thus, there is no guarantee that r_1 and r_2 will enter accept states simultaneously. To overcome this problem, we need to identify the accept states of the intersection of the two automata. To do so, we create two copies of the intersected state space. In the first copy, we check for occurrence of the first acceptance set. In the second copy, we check for occurrence of the second acceptance set. When a run enters a final state in the first copy, we wait for that run also enters in an accept state in the second copy. When this is encountered, we switch back to the first copy and so on. We repeat jumping back and forth between the two copies whenever we find an accepting state.

Definition 2 (Buchi automata intersection). *Let $A_1 = (S_1, s_{10}, F_1, \Sigma, \delta_1)$ and $A_2 = (S_2, s_{20}, F_2, \Sigma, \delta_2)$ be two büchi automata. The intersection $A_1 \cap A_2$ of A_1 and A_2 is the büchi automaton $A = (S, s_0, F, \Sigma, \delta)$ defined as follows:*

- $S = S_1 \times S_2 \times \{1, 2\}$
- $s_0 = (s_{10}, s_{20}, 1)$
- $F = S_1 \times F_2 \times \{2\}$
- *The transition function δ is defined as follows:*

$$\delta((s_1, s_1', 1), p) = \begin{cases} (s_2, s_2', 1) \text{ if } s_2 \in \delta(s_1, p), s_2' \in \delta(s_2, p) \text{ and } s_1 \notin F_1 \\ (s_2, s_2', 2) \text{ if } s_2 \in \delta(s_1, p), s_2' \in \delta(s_2, p) \text{ and } s_1 \in F_1 \end{cases}$$

$$\delta((s_1, s_1', 2), p) = \begin{cases} (s_2, s_2', 2) \text{ if } s_2 \in \delta(s_1, p), s_2' \in \delta(s_2, p) \text{ and } s_1' \notin F_2 \\ (s_2, s_2', 1) \text{ if } s_2 \in \delta(s_1, p), s_2' \in \delta(s_2, p) \text{ and } s_1' \in F_2 \end{cases}$$

Theorem 1. *Let $\psi = \varphi_1 \vee \varphi_2$ (resp. $\psi = \varphi_1 \wedge \varphi_2$) be a LTL formulæ and A_{φ_i} be the büchi automaton equivalent to φ_i for $i = 1, 2$. Let A_ψ be the LTL automaton built according to Definition 1 (resp. Definition 2). Then,* $\mathsf{Words}(\psi) = \mathcal{L}_\omega(A_\psi)$

3. Flat LTL Logic

In this section, we introduce our subset of LTL logic that we call *Flat LTL Logic*. This fragment will be used to express temporal properties and then translate them into Büchi automata in linear size. The syntax of our Flat LTL logic adds to usual boolean propositional operators ¬ (negation) and ∧ (conjunction), some modal operators that describe how the behaviour changes with time.

- **Next**: **Xφ** requires that the formula φ be true in the next state;
- **Until: φ₁ *U* φ₂** requires that the formula φ_1 be true *until* the formula φ_2 is true, which is required to happen;
- **Eventually:** **◊φ** requires that the formula φ be true at some point in the future (starting from the present) and it is equivalent to $\Diamond \varphi \equiv true\ U\ \varphi$;
- **Release: φ₁ *R* φ₂** requires that its second argument φ_2 always be true, a requirement that is *released* as soon as its first argument φ_1 becomes true. It is equivalent to $\varphi_1\ R\ \varphi_2 \equiv \neg(\neg\varphi_1\ U \neg\varphi_2)$.

3.1 Our fragment LTL Logic

Definition 3 (Flat LTL formulæ). *The set of Flat LTL formulæ $\mathcal{L}_f$ is given by the following grammar:*

$$\varphi := \Theta \mid \Theta \mathsf{U} \varphi \mid \varphi \mathsf{R} \Theta \mid \mathsf{X}\varphi \mid \neg\Delta \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

where Θ is a propositional formula defined by the following grammar:

$$\Theta := \mathsf{true} \mid p \mid \neg\Theta \mid \Theta_1 \wedge \Theta_2$$

and Δ is the temporal formula defined by the following grammar:

$$\Delta := \Delta \mathsf{U} \Theta \mid \Theta \mathsf{R} \Delta \mid \mathsf{X}\varphi \mid \neg\Delta \quad \text{with } p \in \Sigma$$

Example: the formula *X*(a *U* ¬(d *R* (¬ b *U* X c))) is not in $\mathcal{L}_f$ since the sub-formula (¬b *U X* c) in ¬(d *R* (¬b *U X* c)) should be of the form Δ *U* θ that is not the case. But, the formula *X*(a *U* ¬ (d *R* (¬ b *R X* c))) is in $\mathcal{L}_f$.

For the sake of brevity and the lack of space, we only discuss here why the fragment θ *U* φ is included within our LTL fragment to the detriment of both formula $\varphi_1 U\ \varphi_2$ and $\varphi_1\ U\ \theta$. It is well-known the size of an Büchi automaton $\overline{A}$ that recognizes the complement language $\mathcal{L}_\omega(\overline{A})$ of the language accepted $\mathcal{L}_\omega(A)$ by an automaton A is exponential [13, 16]. Suppose we have separately built an automaton A_1 for φ_1 and an automaton A_2 for φ_2, and let us then try to compositionally obtain the resulting automaton A for φ. According to the until operator's semantics, it is required that φ holds at the current moment, if there is some future moment for which φ2 holds and φ1 holds at all moments until that future moment. That means constructing the automaton for φ = φ1 U φ2 firstly requires constructing of the intersection of A_1 and $\overline{A_2}$. As stated previously, computing $\overline{A_2}$ is exponential and therefore, constructing the Büchi automaton for $\varphi_1\ U\ \varphi_2$ is exponential. To avoid this kind of formula, we choose the formula θ *U* φ to be a part of our LTL subset where the construction of the Büchi automaton associated to it, does not need to complement any Büchi automaton.

3.2 Flat Positive Normal Form (FPNF)

As LTL formula, flat LTL formula can be transformed into the so-called *Flat Positive Normal form (FPNF)*. This form is characterized by the fact that negations only occur adjacent to atomic propositions. All negation symbols of the given LTL formula have to be pushed inwards over the temporal operators.

Definition 4 (FPNF). *The set of Flat Positive Normal Form (FPNF) formulæ $\mathcal{L}_{FPNF}$ is given by the following grammar:*

$$\varphi := \mathsf{true} \mid \mathsf{false} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Theta \mathsf{U} \varphi \mid \varphi \mathsf{R} \Theta \mid \mathsf{X}\varphi$$

Each formula $\varphi \in \mathscr{L}_f$ can be transformed into a formula $\varphi' \in \mathscr{L}_{FPNF}$. This is done by pushing negations inside, near to atomic propositions. To do this, we use the following transformation rules:

$\neg$ *true* $\rightarrow$ *false*	$\neg(\varphi\ U\ \theta) \rightarrow \neg\varphi\ R\ \neg\theta$
$\neg\neg\varphi \rightarrow \varphi$	$\neg(\varphi_1 \wedge \varphi_2) \rightarrow \neg\varphi_1 \vee \neg\varphi_2$
$\neg X\varphi \rightarrow X\neg\varphi$	$\neg(\theta\ R\ \varphi) \rightarrow \neg\theta\ U\ \neg\varphi$

This transformation is done in linear complexity as it is shown by the following theorem:

Theorem 2. *For any flat LTL formulæ $\varphi \in \mathcal{L}_f$, there exists an equivalent LTL formula $\varphi' \in \mathcal{L}_{FPNF}$ in flat positive normal form with $|\varphi'| = \mathcal{O}(|\varphi|)$.*

Example: the formula $X(a\ U\ \neg(d\ R\ (\neg\ b\ R\ Xc)))$ is in $\mathscr{L}_f$, but not in $\mathscr{L}_{FPNF}$. It can be transformed into $X\ (a\ U\ (\neg d\ U\ (b\ U\ X\ \neg c)))$ which is in $\mathscr{L}_{FPNF}$.

3.3 Semantics

The semantics of LTL formula is defined over infinite[1] sequences $\sigma : \mathbb{N} \rightarrow 2^{\Sigma}$. In other words, a model is an infinite sequence $A_0\ A_1 \cdots$ of subsets of Σ. The function σ, called *interpretation function*, describes how the truth of atomic propositions changes as time progresses. For every sequence σ, we write $\sigma = (\sigma(0), \cdots, \sigma(n), \cdots)$. Thus, we have the following notations:

- **σ(i)** denotes the state at index i and **σ(i:j)** the part of σ containing the sequence of states between i and j;
- **σ(i…)** $= A_i\ A_{i+1}\ A_{i+2} \cdots$ denotes the suffix of a sequence $\sigma = A_0\ A_1\ A_2 \cdots \in (2^{\Sigma})^{\omega}$ starting[2] in the (i+1)st symbol A_i.

We also write $\sigma(i) \vDash \varphi$ to denote that "φ *is true at time instant i in the model* σ". This notion is defined inductively, according to the structure of φ.

The LTL formula are interpreted over infinite sequences of states $\sigma: \mathbb{N} \rightarrow 2^{\Sigma}$ as follows:

Definition 5 (Semantics of FLat LTL). *Let $\sigma : \mathbb{N} \longrightarrow 2^{\Sigma}$ be an interpretation function and $\varphi \in \mathcal{L}$. σ* satisfies *φ, noted $\sigma \vDash \varphi$, is inductively defined over the construction of φ as follows:*

- $\varphi = \mathsf{true}$, *then* $\sigma \vDash \mathsf{true}$
- *if* $\varphi = p$, *then* $\sigma \vDash p$ *iff* $p \in \sigma(0)$
- *if* $\varphi = \mathsf{X}\varphi'$, *then* $\sigma \vDash \mathsf{X}\varphi'$ *iff* $\sigma(1) \vDash \varphi'$
- *if* $\varphi = \Theta \mathsf{U} \varphi$, *then* $\sigma \vDash \Theta \mathsf{U} \varphi$ *iff* $\exists i, i \geq 0, \sigma(i, \ldots) \vDash \varphi$ *and* $\forall j, 0 \leq j < i, \sigma(j...) \vDash \Theta$
- *if* $\varphi = \varphi \mathsf{R} \Theta$, *then* $\sigma \vDash \varphi \mathsf{R} \Theta$ *iff* $\exists i, i \geq 0, \sigma(i, \ldots) \vDash \varphi$ *and* $\forall j, j \geq 0, \sigma(j...) \vDash \Theta$ *or* $\exists i, i \geq 0\ (\sigma(i...) \vDash \varphi \wedge \forall k, k \leq i, \sigma(k...) \vDash \Theta)$
- *if* $\varphi = \neg\varphi'$, *then* $\sigma \vDash \neg\varphi'$ *iff* $\sigma \nvDash \varphi'$
- *Propositional connectives are handled as usual*

The semantics of a LTL formula can be also seen as the language **Words**(φ) that contains all infinite words over the set of atomic propositions (*i.e.* alphabet) 2^{Σ} that satisfy φ. Thus, the language **Words**(φ) for a LTL formula φ is formally defined by **Words**(φ) $= \{\sigma \in (2^{\Sigma})^{\omega} \mid \sigma \vDash \varphi\}$.

Proposition 1. *Two LTL formula φ_1 and φ_2 are* equivalent, *denoted $\varphi_1 \equiv \varphi_2$, if* Words$(\varphi_1)$ =Words(φ_2).

4. Construction Of Buchi Automata For Flat LTL Logic

Our algorithm is a compositional algorithm. It constructs for each sub-formula in our fragment LTL logic, an equivalent Büchi automaton and in a compositional way regroup all resulting Büchi automata in order to get the target Büchi automaton of the target flat LTL formula.

In the sequel, we firstly explain for each sub-formula in our fragment LTL logic how its equivalent Büchi automaton can be obtained.

4.1 Büchi automata for θ formula

The Büchi automaton associated to a propositional formula θ is obtained by creating two states s_0 and s_1 and two transitions tr_1 and tr_2. s_0 is the only initial state while s_1 is the only final state. tr_1 is the transition from s_0 to s_1 labeling with θ while the transition tr_2 is a loop labeled with *true* over the state s_2.

[1] 2^{Σ} is the power set of the proposition set Σ.
[2] ω: is typically used to denote *infinity*.

Definition 6 (Θ automaton). *Let Θ be a propositional formulæ. The automaton $A_\Theta = (S_\Theta, s^0_\Theta, F_\Theta, \Sigma, \delta_\Theta)$ associated to Θ is defined as follows:*

- $S_\Theta = \{s_0, s_1\}$, $s^0_\Theta = s_0$, $F_\Theta = \{s_1\}$
- *The transition function δ is defined as follows:*

$$\delta_\Theta(s_0, \Theta) = \{s_1\} \text{ and } \delta_\Theta(s_1, \text{true}) = \{s_1\}$$

Figure 1 shows the Büchi automaton associated to the propositional formula θ = a ∧ ¬b.

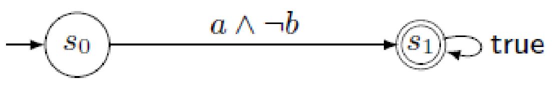


Figure 1: Example of automaton associated to θ

4.2 Büchi automata for θ U φ formula

The main idea behind the composition θ *U* φ is to add a new initial (nonaccept) state s_{new} to the set of states of the automaton A_φ associated to φ with the following transitions:

a) A loop over the added state s_{new} labelled with the propositional formula θ

b) Transitions s_{new} to a state s labelled with a proposition p if and only if there a transition from the initial state s^0 of A_φ to the state s labelled with the proposition p .

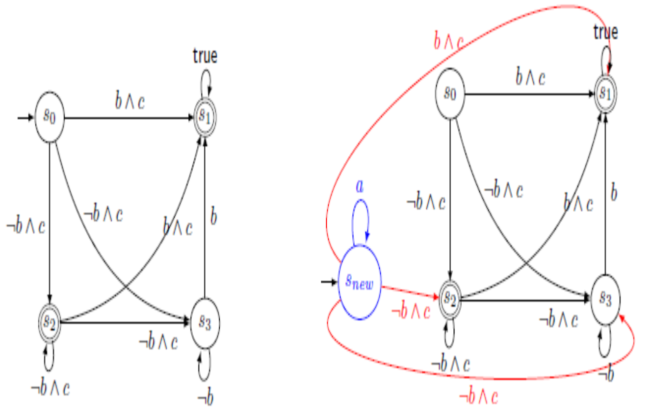
All other transitions of A_φ, as well as the accept states, remain unchanged. The state s_{new} is the single initial state of the resulting automaton, is not accept, and, clearly, has no incoming transitions except the loop one.

Definition 7 (Θ U φ automaton). *Let Θ be a propositional formula and φ be an LTL flat formulæ. Let $A_\varphi = (S_\varphi, s^0_\varphi, F_\varphi, \Sigma, \delta_\varphi)$ be the automaton associated to φ. The automaton $A_\psi = (S_\psi, s^0_\psi, F_\psi, \Sigma, \delta_\psi)$ associated to $\psi = \Theta \cup \varphi$ is defined as follows:*

- $S_\psi = \{s_{\text{new}}\} \cup S_\varphi$
- $s^0_\psi = s_{\text{new}}$, $F_\psi = F_\varphi$
- *The transition function δ_ψ is defined as follows:*

$$\delta_\psi(s,p) = \begin{cases} \delta_\varphi(s,p) \text{ if } s \in S_\varphi \ (A_\varphi \text{ transitions}) \\ \delta_\varphi(s^0_\varphi, p) \text{ if } s = s_{\text{new}} \ (\text{Connection initial state to } A_\varphi) \\ \{s_{\text{new}}\} \text{ if } s = s_{\text{new}} \text{ and } p = \Theta \ (\text{Loop over the new initial state}) \end{cases}$$

Example: Figure 2 illustrates the composition definition of θ *U* φ. **Figure 2a** shows the Büchi automaton associated to (◊ b) *R* c. To construct the Büchi automaton associated to (a *U* ((◊ b) *R* c)), we add a new state *snew* that we consider as initial state. Then, for each transition outgoing from snew with label l and goes to state s, we add a transition from *snew* to the state s with a label l. Finally, we then add a loop labeled with the atomic proposition a over the added state.



(a) (◊b) R c (b) a U (◊b R c)

Figure 2: Example of composition: θ *U* φ

Theorem 3. *Let $\psi = \Theta \cup \varphi$ be a flat LTL formulæ and A_φ be the büchi automaton equivalent to φ. Let A_ψ be the automaton built according to Definition 7. Then, $\text{Words}(\psi) = \mathcal{L}_\omega(A_\psi)$.*

4.3 Eventually operator ◊φ:

The Büchi automaton construction of the formula ◊φ is a particular case of the Büchi automaton construction of the formula θ *U* φ where θ is the *true* formula. Thus, the main idea behind the composition ◊φ is to add a new initial (nonaccept) state s_{new} to the automaton states set A_φ associated to φ with the same transitions defined for θ *U* φ where the loop over the added state s_{new} is labeled with *true* instead of the atomic formula θ.

4.4 Büchi automata for Xφ formula

The main idea behind the composition *X*φ consists in adding two new states s_{new} (neither initial state or accept state) and s_{init} (considered as the initial state) to the state set of the automaton A_φ with the following transitions:

a) Add for any transition in A_φ which starts from the initial state s^0 to a state s, a transition from s_{new} to s;

b) Add a transition from the initial state s_{init} to the s_{new} labeled with *true*.

All other transitions of A_φ remain unchanged and final states of A_φ become accept ones of A_ψ and initial state of A_ψ become the state s_{init}.

Definition 8 ($X\varphi$ **automaton**). *Let φ be an Flat LTL formulæ. Let $A_\varphi = (S_\varphi, s^0_\varphi, F_\varphi, \Sigma, \delta_\varphi)$ be the automaton equivalent to φ. The* **automaton** $A_\psi = (S_\psi, s^0_\psi, F_\psi, \Sigma, \delta_\psi)$ *equivalent to $\psi = X\varphi$ is defined as follows:*

- $S_\psi = S_\varphi \cup \{s_{\mathsf{new}}, s_{\mathsf{init}}\}$
- $s^0_\psi = s_{\mathsf{init}}$, $F_\psi = F_\varphi$
- *The transition function δ is defined as follows:*

$$\delta_\psi(s,p) = \begin{cases} \delta_\varphi(s,p) \text{ if } s \in S_\varphi \ (A_\varphi \text{ transitions}) \\ \delta_\varphi(s^0_\varphi,p) \text{ if } s = s_{\mathsf{new}} \ (\text{Connection } s_{\mathsf{new}} \text{ state to initial state of } A_\varphi) \\ \{s_{\mathsf{new}}\} \text{ if } s = s_{\mathsf{init}} \text{ and } p = \mathsf{true} \ (\text{Connection } s_{\mathsf{init}} \text{ to } s_{\mathsf{new}}) \end{cases}$$

Example: Figure 3 illustrates the definition of $X\varphi$. **Figure 3a** shows the Büchi automaton associated to the formula (a U (X b R c)). To construct the Büchi automaton equivalent to X(a U (Xb R c)), we add a new state s_{new} and for each transition tr starting from the initial state s^0_φ to a state s, a transition from s_{new} to s with the same label. Finally, we add the state s_{init} that we consider as initial and we connect s_{init} to s_{new} with a transition labeled with the *true* label.

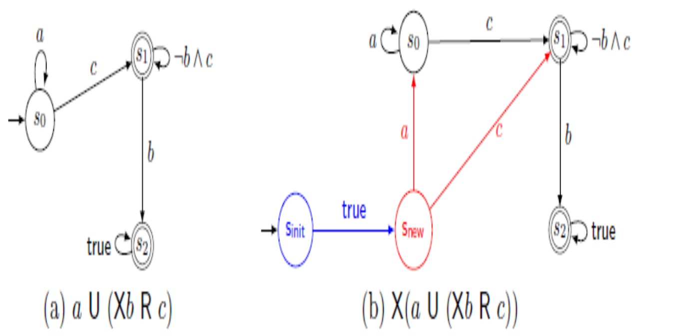


Figure 3: Example of composition: $X\varphi$

Theorem 4. *Let $\psi = X\varphi$ be a LTL formulæ and A_φ be the büchi automaton equivalent to φ. Let A_ψ be the LTL automaton built according to Definition 8. Then,* $\mathsf{Words}(X\varphi) = \mathcal{L}_\omega(A_\psi)$.

4.5 Büchi automata for φ R θ formula

The formula φ R θ informally means that θ is true until φ becomes true, or θ is true forever. Thus, the construction of a Büchi automaton for φ R θ can be done by construction the Büchi automaton associated to the fact that θ is true until φ becomes true and the construction of a Büchi automaton associated to the fact that θ is true forever. Finally, make the union between the two constructed Büchi automata. Consequently, to build the Büchi automaton for φ R θ, we need to add two new states s_i and s_f to the set of states of the automaton A_φ. s_i becomes the single initial state of the resulting automaton and s_f is added to set of final states of the resulting automaton. The following transitions are added to the set of transitions of the resulting automaton:

a) For any transition from the initial state s^0 of A_φ to a state s labeled with a proposition p, add a transition from the state s_i to s labeled with the proposition p ∧ θ;

b) A loop over the added state s_i labeled with the propositional formula θ;

c) A loop over the added state s_f labeled with the propositional formula θ;

d) A transition from the state s_i to the state s_f labeled with the proposition θ.

All other transitions of A_φ, as well as the accept states, remain unchanged.

Definition 9 (φ R Θ **automaton**). *Let Θ be a propositional formula and φ be an LTL flat formulæ. Let $A_\varphi = (S_\varphi, s^0_\varphi, F_\varphi, \Sigma, \delta_\varphi)$ be the automaton associated to φ. The* **automaton** $A_\psi = (S_\psi, s^0_\psi, F_\psi, \Sigma, \delta_\psi)$ *associated to $\psi = \varphi$ R Θ is defined as follows:*

- $S_\psi = \{s_i, s_f\} \cup S_\varphi$
- $s^0_\psi = s_i$, $F_\psi = F_\varphi \cup \{s_f\}$
- *The transition function δ is defined as follows:*

$$\delta_\psi(s,p) = \begin{cases} \delta_\varphi(s,p) \text{ if } s \in S_\varphi \ (A_\varphi \text{ transitions}) \\ \delta_\varphi(s^0_\varphi,p\prime) \text{ if } s = s_i \text{ and} \\ \qquad p = \Theta \wedge p\prime \ (\text{Connection } s_i \text{ to initial state of } A_\varphi) \\ \{s_i, s_f\} \text{ if } s = s_i \text{ and } p = \Theta \ (\text{Loop over } s_i \text{ or connection } s_i \text{ to } s_f) \\ \{s_f\} \text{ if } s = s_f \text{ and } p = \Theta \ (\text{Loop over } s_f) \end{cases}$$

Example: Figure 4 illustrates the composition definition of φ R θ. **Figure 4a** shows the Büchi automaton associated to the formula c U ◊b. To construct the Büchi automaton associated to the flat LTL formula ((c U ◊b) R a), we add a state s_i that we consider as the only initial state and a state s_f that we consider as a final state. We add a loop labelled with the atomic proposition a over the two added states. Finally, for each transition outgoing from the initial state of the automaton φ with label l and goes to state s, we add a transition from the added state s_i to the state s with a label (l ∧ a). We also add a transition labelled with a from the state s_i to the state s_f.

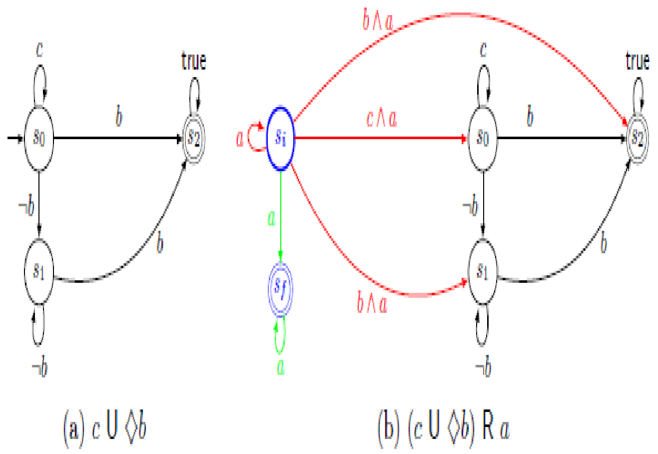


Figure 4: Example of composition: φ *R θ*

Theorem 5. *Let $\psi = \varphi \; \mathrm{R} \; \Theta$ be a LTL formulæ and A_φ be the büchi automaton equivalent to φ. Let A_ψ be the LTL automaton built according to Definition 9. Then,* $\mathrm{Words}(\varphi \; \mathrm{R} \; \Theta) = \mathcal{L}_\omega(A_\psi)$.

5. Finite syntax tree of flat LTL formula

A flat LTL formula φ can be transformed into a tree containing all the information about the possible sub-formula of φ. It will form the cornerstone of the construction of Büchi automata from flat LTL formula. We assume that our flat LTL formula are fully parenthesized and we show how to build the finite syntax tree, named *FST*(φ), algorithmically for a flat LTL formula φ. This tree can be thought of as a data structure representing the sub-formula after a finite breaking up the formula into a list of tokens. We distinguish four kinds of tokens: left brackets "(", right brackets ")", FLTL operators and propositional variables. FLTL operators represent the internal nodes of our tree while the propositional variables represent the leaf nodes. Our algorithm to build *FST*(φ) is[3] inspired from [5] and uses a stack for operators and a stack for propositional variables, and consists of the following rules:

a) If the current token is a left bracket "(" (*i.e.* we are reading a new sub-formula), push it on the operator stack;

b) If the current token is a operator (*i.e.* in {'∧', '∨', '*X*', '*U*', '◊', '*R*', '¬' }), push it on the operator stack;

c) If the current token is a propositional variable p, create a tree with single node whose the value is p and push the created tree on the variable stack;

d) If the current token is a right bracket ")" (*i.e.* we have just finished reading a sub-formula), pop operators off the operator stack while this operator is not a left bracket. If the popped operator is an unary operator, pop one tree variable from variable stack and create new tree whose the root is the popped operator and it is only child is the popped tree. If the popped operator is a binary operator, pop two tree variables from variable stack and create new tree whose the root is the popped operator and its right child the first popped tree and its left child the second popped tree. If no left bracket is found during popping the variable stack, throw a mismatched bracket expression. Otherwise, pop found left bracket from the operator stack;

e) At the end of reading expression tokens, pop all operators off the operator stack and for each popped operator:

- If the popped operator is an unary operator, pop one tree variable from variable stack and create new tree whose the root is the popped operator and it is only child is the popped tree. Then, push the created tree on the variable stack;

- If the popped operator is a binary operator, pop two tree variables from variable stack and create new tree whose the root is the popped operator and its right child the first popped tree and its left child the second popped tree. Then, push the created tree on the variable stack;

- If the popped operator is left or right bracket, throw an unbalanced left brackets.

Hence, our mechanism of creating *FST*(φ) can be described by the algorithm illustrated in **Figure 5.**

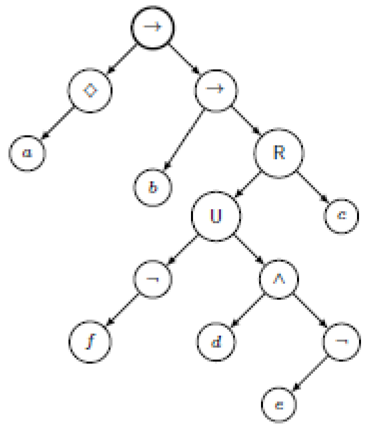
[3] Shunting-yard algorithm proposed by *Djikstra* and used to parse mathematical expressions specified in infix notation.

```
Input: a positive flat LTL formulæ φ  Output: the finite syntax tree
 FST(φ);
operatorStack ← createStack();
operandStack ← createTreeStack();
l ← split(φ);
for e ∈ l do
    if isSpace(e) then
        continue;
    else if leftBracket(e) or unary(e) or binary(e) then
        push(operatorStack, e);
    else if variable(e) then
        push (operandStack,createNode (e));
    else if rightBracket(e) then
        while !emptyStack (operatorStack) do
            popped ← pop (operatorStack);
            if unary (popped) then
                push (operandStack,addRight
                 (popped,pop(operandStack)));
            else if binary(popped) then
                push(stackOperand,addRightLeft (popped,pop
                 (stackOperand),pop (stackOperand),e);
            else
                break; //encountered a left bracket
        end
        if emptyStack (operatorStack) then
            throw Exception("Unbalanced right parentheses");
    else
        throw Exception(Unknown token);
end
while !emptyStack(operatorStack) do
    popped ← top (operatorStack);
    pop (operatorStack);
    if unary (popped) then
        push (operandStack,addRight (popped,pop(operandStack)));
    else if binary(popped) then
        push(stackOperand,addRightLeft (popped,pop
         (stackOperand),pop (stackOperand),e);
    else
        throw Exception("Unbalanced left parentheses");
end
if lenght (operandStack)=1 then
    return top (operandStack);
else
    throw Exception("Error in LTL expression");
```

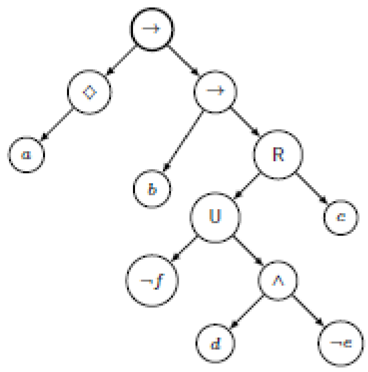
Figure 5: Building syntax tree for a FLTL formula

Example: Figure 6a shows the finite syntax tree *FST*(φ) generated for the FLTL expression:

$$\varphi = \Diamond\, a \rightarrow (b \rightarrow ((\neg f)\ U\ (d \wedge (\neg e))))\ R\ c.$$



(a) $FST(\varphi)$



(b) $FST(\varphi)$ with negations are pushed to leaves

Figure 6: Example of finite syntax tree

This finite syntax tree will be used to construct the Büchi automaton equivalent to a flat LTL formula φ in flat positive normal form. Since our algorithm takes as input a flat positive LTL formula, any node in the finite syntax tree labeled with the negation operator ¬ is certainly located directly before a leaf. For technical reasons, we merge the nodes labeled with ¬ with the leaf which directly follows in the finite syntax tree. **Figure 6b** illustrates the finite syntax tree presented in **Figure 6a** after pushing negations to leaves.

6. FROM FINITE SYNTAX TREE TO BUCHI AUTOMATA

Our algorithm to build Büchi automata from flat LTL formula is compositional in the sense that the final Büchi automaton is obtained by developing a sub-automaton for each sub-formula of the principal formula. Hence, the basic idea for developing the final automaton for a flat LTL formula φ is to explore *FST*(φ) in a pre-order traversal. That is to say, we visit the root node first, then recursively do a pre-order traversal of the left sub-tree, followed by a recursive pre-order traversal of the right sub-tree. The algorithm, illustrated in **Figure 7,** allows us to build a Büchi automaton from a finite syntax tree of a positive flat LTL formula T=*FST*(φ) and uses the following five functions:

a) **CreateBuchiProp(θ):** takes as input a propositional formula θ and returns the automaton as defined in Definition 6 (Section 4);

b) **CreateBuchiUnary(op, BA)**: takes as input an unary LTL formula (*i.e.* op ∈ {*X*, ◊}) and a Büchi automaton BA and returns a Büchi automaton defined according to definitions of ◊ and *X* given in Section 4;

c) **CreateBuchiBinary(op, BA_l,BA_r)**: that takes as input ∧ or ∨ operator and two Büchi automata BA_l and BA_r and returns a Büchi automaton defined according to definitions of ∧ and ∨ given in Section 2;

d) **BuchiUntil(θ, BA)**: that takes as input a propositional formula θ and a Büchi automaton BA and returns the automaton as defined in Definition 7 (Section 4);

e) **BuchiRelease(θ, BA):** that takes as input a propositional formula θ and a Büchi automaton BA and returns the automaton as defined in Definition 9 (Section 4).

```
Name : BuildBA
Input  : a finite syntax tree in which negations are pushed to
         leaves  T = FST(φ)
Output: a büchi automaton A
A_φ ← CreateEmptyBA();
if IsEmpty(T) then
 | return CreateEmptyBA();
else if IsLeaf(T) then
 | return CreateBuchiProp(Root(T));
else
 | if Unary(Root(T)) then
 | | return CreateBuchiUnary(Root(T), BuildBA(Left(T)));
 | else if Until(Root(T)) then
 | | return BuchiUntil(Root(Left(T)), BuildBA(Right(T)));
 | else if Release(Root(T)) then
 | | return BuchiRelease(Root(Right(T)), BuildBA(Left(T)));
 | else
 | | return CreateBuchiBinary(Root(T),BuildBA(Left(T)),
 | |  BuildBA(Right(T)));
end
```

Figure 7: building buchi automata: buildBA(T)

Theorem 6. *For any flat LTL formulæ $\varphi \in \mathcal{L}_f$, there exists an büchi automaton A_φ with $|A_\varphi| = O(|\varphi|)$.*

Theorem 7. *Let $FST(\psi)$ be the finite syntax tree of a flat LTL formulæ ψ and A_ψ is the büchi automaton generated by Algorithm 2, then: $Words(\psi) = \mathcal{L}_\omega(A_\psi)$*

7. CONCLUSION AND FUTURE WORK

This paper presents a compositional algorithm for generating Büchi automata from a fragment of LTL logic. We firstly proposed the grammar of this fragment and then built for each formula φ, its equivalent automata. We secondly showed how to compositionally build from Büchi automata associated to each sub-formula, the Büchi automaton of the target formula. We thirdly showed the complexity and the correctness of our Büchi automata generation method.

Future work: several research lines can be continued from the present work. First, some temporal operators such as always, precedes or since are not considered in this paper, as an immediate perspective, we will study how to include these operators in our LTL fragment. Second, in [6, 7], *Dwyer*'s presents a translational semantics for his pattern properties. Indeed, for each pattern property, he associates an equivalent LTL formula. In [17], the authors show how Büchi automata can be polynomially generated from pattern properties proposed by *Dwyer*. It will be interesting to study whether the translational semantics given by *Dwyer* is covered by our fragment. This will be done by comparing Büchi automata generated by the algorithm proposed in [17] with the Büchi automata generated by our algorithm.

REFERENCES

[1] C. Baier and J.P. Katoen. Principles of Model Checking (Representation and Mind Series). The MIT Press, 2008.

[2] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In Formal methods in system design, pages 415-427. Springer-Verlag, 1994.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., 8(2):244-263, April 1986.

[4] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, Cambridge, MA, USA, 1999.

[5] E.W. Dijkstra. An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical Report 35, Mathematisch Centrum, Amsterdam, 1961.

[6] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In FMSP, pages 7- 15, 1998.

[7] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In Proceedings of the 21st International Conference on Software Programming, pages 411 - 420, 1999.

[8] P. Gastin and D. Oddoux. Fast LTL to Buchi automata translation. In Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), volume 2102 of LNCS, pages 53- 65, Paris, France, jully 2001. Springer.

[9] V. King, O. Kupferman, and M.Y. Vardi. On the Complexity of Parity Word Automata, pages 276 -286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[10] M. Mukund. Finite-state automata on infinite inputs, 1996.

[11] A. Pnueli. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77, pages 46-57, Washington, DC, USA, 1977. IEEE Computer Society.

[12] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Proceedings of the 5th Colloquium on International Symposium on Programming, pages 337 - 351, London, UK, UK, 1982. Springer-Verlag.

[13] S. Safra. On the complexity of omega-automata. In 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988, pages 319- 327, 1988.

[14] S. Safra. Complexity of Automata on Infinite Objects. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, March 1989.

[15] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. J. ACM, 32(3):733- 749, july 1985.

[16] A.P Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. In Automata, Languages and Programming, pages 465 - 474, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[17] S. Taha, J. Julliand, F. Dadeau, K. Castillos, and B. Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. Formal Asp. Comput., 27(4):641 - 664, 2015.

[18] M. Y. Vardi and P.Wolper. An automata-theoretic approach to automatic program verification. In Proc. 1st Symp. on Logic in Computer Science, pages 332 - 344, Cambridge, June 1986.

[19] M.Y. Vardi. Branching vs. linear time: Final showdown. In Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001, pages 1 - 22, London, UK, 2001. Springer-Verlag.