# LTLf Synthesis - Technical Report

**Rémy LE BOHEC**
(supervisor: Philipp Schlehuber-Caissier)

Technical Report $n^o$202407-techrep-le-bohec, July 2024
revision 159e317

We address the problem of synthesis and model checking given a LTL specification over finite traces. Our contributions to this topic are two-fold. First, relying on the open source tool spot Duret-Lutz et al. (2022), we implement algorithms allowing for a direct translation from a LTLf formula to a finite automaton. Then, we use this algorithm to perform model checking by building on the fly the automaton associated with the specification.

Nous nous intéressons au problème de la synthèse et de la vérification de modèle selon une spécification LTL sur des traces finies. Nos contributions à ce sujet sont de deux ordres. Dans un premier temps, à l'aide de l'outil open source spot Duret-Lutz et al. (2022), nous souhaitons implémenter des algorithmes permettant la transformation directe d'une formule LTLf en un automate fini. Par la suite, nous souhaitons pouvoir effectuer la vérification d'un controlleur en construisant à la volée l'automate associé à la spécification.

**Keywords**
LTLf, Synthesis, Model checking, On the fly


Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
remy.le-bohec@epita.fr – http://www.lre.epita.fr/

## Copying this document

# Contents

# Motivations

Linear Temporal Logic has become a staple in the field of methods for specifying the temporal properties of systems. Its ability to describe evolutions in the states of a system has made it crucial for both the specification and verification of hardware and software alike.

It is extensively used in the verification of reactive systems: systems which continuously react to inputs in their environment. It allows us to express properties of safety and liveness (to ensure nothing bad ever happens and something good eventually happens).

It is also applicable to various other domains, notably business process specification and verification, where the operation is finite (with a precise beginning and end).

The idea of studying Linear Temporal Logic over Finite Traces finds purpose in other domains such as Artificial Intelligence where it is used to specify goals and constraints in automated planning, formalizing temporal objectives. In robotics, it can also be useful to ensure systems operate safely and predictably within their environments.

# Chapter 1

# Concepts

## 1.1 Linear temporal logic

Linear Temporal Logic (LTL) was introduced in Pnueli (1977) as a formalism used in formal specification and verification to describe and reason about sequences of Boolean variables over time. It extends classical Boolean logic with temporal operators. LTL is used to specify correct system behavior (the LTL formula is precisely the rule that needs to be followed).

LTL formulas are used to verify if the behavior of a system follows a given set of rules or specifications. The verification process ensures that systems operate correctly and reliably, "proving" the functionality of the code.

A system possesses a set of Boolean variables known as atomic propositions. These propositions correspond to a single observable state or event within the model or the system. For instance, an atomic proposition might represent if a sensor detects a person in a room, whether a light is on or off, if a network packet has been received or not.

At any given instant, the studied system is in a specific configuration, which corresponds to the truth values of all atomic propositions at that specific time. Each configuration can be considered a state of the system itself.

In LTL, the behavior of a system, also referred to as a trace, is an infinite sequence of configurations. The notion of infinite traces models the continuous execution of systems over time.

In the following sections, we will look into the syntax and semantics of LTL to understand how formulas are constructed and interpreted.

### 1.1.1 Syntax

LTL formulas can be constructed in two ways.

First, by combining other LTL formulas with Boolean operators:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \implies \varphi$$

$\varphi ::= p$:
A formula can be a simple atomic proposition $p$, in which case $p$ must be true on the current step. The negation $\neg$, conjunction $\wedge$, disjunction $\vee$ and implication work for temporal formulas as they do in Boolean logic.

In order to reason over the evolution of configurations of the system, LTL introduces several temporal operators:

$$\varphi ::= X\varphi \mid F\varphi \mid G\varphi \mid \varphi U\varphi \mid \varphi W\varphi \mid \varphi R\varphi \mid \varphi M\varphi$$

$\varphi ::= X\varphi$:
$X$ is the next operator. It means that the formula it is applied to must hold on the next step, at $t + 1$.

$\varphi ::= \varphi_1 U \varphi_2$:
$U$ is the strong until operator. The formula $\varphi_1$ must be true until the formula $\varphi_2$ becomes true, and the formula $\varphi_2$ must eventually become true.

The following formulas can be considered syntactic sugar, as we can build them using only the aforementioned Boolean, next and until operators:

$\varphi ::= G\varphi$:
$G$ is the 'globally' (or always) operator. It means that the formula it is applied to must hold from the current step onward.

$\varphi ::= F\varphi$:
$F$ is the 'finally' (or eventually) operator. It means that the formula it is applied to must hold on the current step or at some point in the future.

$\varphi ::= \varphi_1 W \varphi_2$:
$W$ is the weak until operator. The formula $\varphi_1$ must be true until the formula $\varphi_2$ becomes true, and the formula $\varphi_2$ may never become true, in which case the formula $\varphi_1$ must hold forever inwards.

$\varphi ::= \varphi_1 R \varphi_2$:
$R$ is the weak release operator. The formula $\varphi_2$ must be true until the formula $\varphi_1 \wedge \varphi_2$ becomes true. $\varphi_1 \wedge \varphi_2$ may never become true, in which case $\varphi_2$ must hold forever inwards.

$\varphi ::= \varphi_1 M \varphi_2$:
$M$ is the strong release operator. The formula $\varphi_2$ must be true until the formula $\varphi_1 \wedge \varphi_2$ becomes true. $\varphi_1 \wedge \varphi_2$ must become true at some point in the future.

### 1.1.2 Semantics

The semantics of LTL are defined over traces, which represent the evolution of a system's states over time.

A **trace** is an infinite sequence of assignments to the atomic propositions.

Given a trace $\sigma = s_0, s_1, s_2, \ldots$, we can define when a trace models an LTL formula using the satisfaction relation $\models$.

Formally, the satisfaction relation $\sigma, i \models \varphi$ means that the trace $\sigma$ starting from position $i$ satisfies the formula $\varphi$. The definition is as follows:

$\sigma, i \models p$ if $p$ is true in $s_i$.

$\sigma, i \models \neg\varphi$ if $\sigma, i \not\models \varphi$.

$\sigma, i \models \varphi_1 \vee \varphi_2$ if $\sigma, i \models \varphi_1$ or $\sigma, i \models \varphi_2$.

$\sigma, i \models \varphi_1 \wedge \varphi_2$ if $\sigma, i \models \varphi_1$ and $\sigma, i \models \varphi_2$.

$\sigma, i \models \varphi_1 \implies \varphi_2$ if $\sigma, i \not\models \varphi_1$ or $\sigma, i \models \varphi_2$.

$\sigma, i \models X\varphi$ if $\sigma, i+1 \models \varphi$.

$\sigma, i \models F\varphi$ if there exists some $j \geq i$ such that $\sigma, j \models \varphi$.

$\sigma, i \models G\varphi$ if for all $j \geq i$, $\sigma, j \models \varphi$.

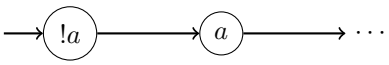
$\sigma, i \models \varphi_1 U \varphi_2$ if there exists some $j \geq i$ such that $\sigma, j \models \varphi_2$ and for all $k$ such that $i \leq k < j$, $\sigma, k \models \varphi_1$.

$\sigma, i \models \varphi_1 W \varphi_2$ if $\sigma, i \models \varphi_1 U \varphi_2$ or $\sigma, i \models G\varphi_1$.

$\sigma, i \models \varphi_1 R \varphi_2$ if for all $j \geq i$, $\sigma, j \models \varphi_2$ or there exists some $k$ such that $i \leq k < j$ and $\sigma, k \models \varphi_1 \wedge \varphi_2$.

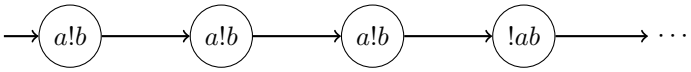
$\sigma, i \models \varphi_1 M \varphi_2$ if for all $j \geq i$, $\sigma, j \models \varphi_2$ and there exists some $k \geq i$ such that $\sigma, k \models \varphi_1 \wedge \varphi_2$.

**Example:**
For the formula $\varphi = Xa$, where $a$ is an atomic proposition, an accepting trace could be:




On the first state, we do not actually need to check for anything, because the next operator applies on the second state. However, on the second state, we need $a$ to be true. All following states can be of any configuration.

**Example:**
For the formula $a\ U\ b$, where $a$ and $b$ are atomic propositions, an accepting trace could be:




Let us start from the first state at which $t = 0$.
On this state, $a$ is true, and $b$ is false. Since $b$ isn't true on this state, $a$ must be true and the formula must hold on the next state.
On the state at which $t = 1$ and $t = 2$, we have the same configuration, therefore we need the formula to hold on the next state as well.
At $t = 3$, $b$ is true. Therefore, $a$ no longer needs to hold, and the formula is verified on the current trace. Subsequent states can be of any configuration because we have already verified the formula on the trace.

## 1.2 Linear temporal logic over finite traces

LTLf De Giacomo and Vardi (2013) is a modification of LTL intended for reasoning about finite sequences of states. Unlike LTL which assume traces to be infinite, LTLf is better-suited for scenarios where runs are bounded in time, such as business processes or workflows.

### 1.2.1 Syntax

The syntax of LTLf is similar to that of LTL, having the same set of temporal operators except for next which is split between weak next ($X$) and strong next ($X[!]$).

$\varphi ::= X\varphi$: The weak next operator is used to say that the formula $\varphi$ must hold on the next state if it exists.

$\varphi ::= X[!]\varphi$: On the other hand, the strong next operator requires a following state to exist and for the formula $\varphi$ to hold at this state.

In the context of infinite traces, the next operator cannot be weak since every state of the trace has a successor.

### 1.2.2 Differences with LTL

While both logics seem to be quasi-identical, LTLf allows for some simplifications or even cases where the accepted traces between one and the other greatly differ.

**Example:** Consider the formula $\varphi = G(F\ p)$, also known as the liveness property.

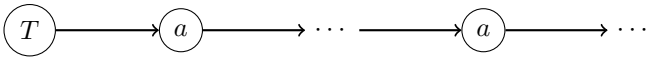
In LTL, this means that the atomic proposition $p$ must be true an infinite number of times: for every step in which $p$ holds, a future step in which $p$ holds as well is guaranteed to exist.

However, in LTLf, this simply means that the last state of the trace must be in a configuration where $p$ is true. Indeed, we need $p$ to eventually be true on all states of the trace, which is equivalent to ending the execution on $p$.

**Example:**
Consider the formula $\varphi = G(X[!]p)$, where $X[!]$ is the strong next operator for LTLf, and the regular next operator for LTL.

In the case of LTL, the formula is equivalent to $X(Gp)$, because the system can be in any configuration on the first state but $p$ must hold on every state thereafter.

A valid trace could have been of the form:




Where $a$ must hold on every state starting from the second.

On the other hand, in LTLf, the formula is not equivalent to $X(Gp)$. The globally operator applied to the strong next means that for each state in the trace, we need the following state to exist and the property $p$ to hold in it. However, this requires for each state in the trace to have another state after it, causing all finite traces to be rejected.

### 1.2.3 Motivations for studying LTLf

Many real-world systems, such as processes, workflows or planning algorithms, operate within a finite window of time compared to systems which should run indefinitely, such as operating systems. For those systems, a logic restricted to finite traces is more appropriate.

Furthermore, LTLf formulas can be translated into simpler automata than LTL. Indeed, we do not need Büchi acceptance for infinite words when working on finite traces: we can simply build finite automata.

Formula are convenient for specification but have no canonical form and are complicated to work with algorithmically. There, to manipulate them or use them for verification, we first translate them to automata as proposed in the next section.

### 1.2.4 Related works

Many published articles have used LTL to reason over finite traces in the past. For instance, the use of temporal logic to specify complex goal criteria in automated planning Bacchus and Kabanza (1998). LTL is used to help connect extended goals into planning algorithms.

Consequently, LTLf was introduced in 2013 De Giacomo and Vardi (2013) to offer a better suited

framework for finite executions. It was demonstrated that LTLf retains much of the expressive power of LTL and could be used for the study of finite systems or in planning algorithms Camacho et al. (2018).

The framework of synthesis was then extended to finite traces De Giacomo and Vardi (2015) where automata theoretic techniques were reused for LTLf formulas. This advancement enabled the generation of controllers better suited for finite executions.

Studies were also done on LTLf formulas considered insensitive to infiniteness De Giacomo et al. (2014): formulas that could correctly be manipulated as LTL formulas (thus on infinite traces), leading to blurring between LTL and LTLf.

The idea of directly translating LTLf formulas into deterministic finite automata rather than nondeterministic De Giacomo and Favorito (2021) was also approached by inductively transforming each sub-formula into a DFA and combining them all using automata operators in a scalable and practical manner.

Other approaches to LTLf synthesis have been conducted, including symbolic LTLf synthesis Zhu et al. (2017) in order to apply mechanisms of Boolean synthesis to generate strategies. In the following chapters, we will build upon this idea of symbolic automata to perform verification.

# Chapter 2

# Translation

In order to work with formulas easily and efficiently, automata theoretic approaches to LTL Vardi (1996) were introduced which enabled the use of automata on infinite words to analyze and manipulate formulas.

We now turn our attention to the process of translating LTLf formulas into deterministic and nondeterministic finite automata (DFA and NFA). These FA offer the use of algorithmic techniques for verification similar to the acceptance of a word, where words are now traces and letters are configurations.

**Definition: (Deterministic Finite Automaton):** A Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of:

- a finite set of states $Q$
- a finite set of input symbols called the alphabet $\Sigma$
- a transition function $\delta : Q \times \Sigma \to Q$
- an initial state $q_0$
- a set of accepting states $F \subset Q$

**Definition: (Nondeterministic Finite Automaton):** A Nondeterministic Finite Automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of:

- a finite set of states $Q$
- a finite set of input symbols called the alphabet $\Sigma$
- a transition function $\delta : Q \times \Sigma \to P(Q)$, the power set of $Q$
- an initial state $q_0$
- a set of accepting states $F \subset Q$

The transition function $\delta$ maps a pair of $(q_i, a)$ to a set of states $S \subset Q$, meaning that reading the input $a$ on state $q_i$ can set the system to any state $q_j \in S$.

To check whether a FA accepts a word, we start on the initial state and for each letter of the

word, we take the transition given by the transition function. A word is said to be accepted if the last state we reach is included in the accepting states $F$ or, in the case of NFA, if the set of states we reach last contains at least one final state.

## 2.1 Semi-symbolic automata

In our case, the we would like to study whether traces representing an execution of a system verify a given specification in LTLf. Our alphabet is the minterms set $2^{AP}$, where $AP$ is the set of atomic propositions.

The automata we are working with are semi-symbolic automata: instead of accepting letters, transitions are labeled using binary decision diagrams (BDDs), rooted and directed acyclic graphs encoding functions of Boolean variables.

Having BDDs on the transitions allows us to encode more efficiently parallel transitions by regrouping them with a single BDD representing the set of all minterms that reach a given formula.

As a result, given the atomic propositions $p$ and $q$, instead of having one edge per minterm, we can transitions with BDDs corresponding to formulas such as $p \vee q$ instead of $p \wedge q, p \wedge \neg q, \neg p \wedge q$ or $p$ instead of $p \wedge q$ and $p \wedge \neg q$.

Unless specified otherwise, the automata used in the following sections are semi-symbolic finite automata.

## 2.2 Formula rewriting

On a given state of a FA, we can verify a condition that occurs at the current time $t$. Anything that has to be verified in the future is propagated to the further states of the automaton. Therefore, when translating a LTLf formula $\varphi$, we can rewrite the formula as $c \wedge X\psi$ where c is a Boolean formula that can be evaluated on the current state, and $\psi$ the formula that should be verified in a future state. In order to simplify rewritings, we also introduce ways to rewrite Boolean and temporal operators with other operators.

The rules for rewriting formulas are as follows:

$\bigwedge_{i=0}^{k} \varphi_i$: we split the conjunction up and add a new state for each $\varphi_i$, while combining all immediate verifications in the current state.

$\bigvee_{i=0}^{k} \varphi_i$: similarly to Thompson's construction algorithm, we make branchings transitioning to each formula, without the use of epsilon transitions.

$\varphi \implies \psi$: this formula is simply rewritten to $\neg\varphi \vee \varphi \wedge \psi$.

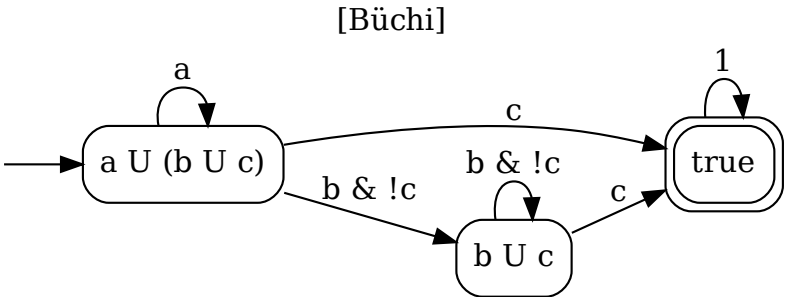



Figure 2.1: NFA for $a\ U\ (b\ U\ c)$

$\varphi \iff \psi$: this formula is simply rewritten to $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$.

$\varphi \oplus \psi$: this formula is simply rewritten to $(\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi)$.

$F\ \varphi$: Finally means that the formula should be accepted now or at some point in the future: therefore, we can rewrite it to $\varphi \vee X(F\ \varphi)$.

$G\ \varphi$: Globally of $\varphi$ means that we want to verify $\varphi$ now and on the next state to keep verifying $G\ \varphi$: we obtain $\varphi \wedge X(G\ \varphi)$.

$\varphi\ U\ \psi$: we either verify $\psi$ or need to verify $\varphi$ and keep checking for $\varphi\ U\ \psi$ on the next state: the formula is thus rewritten as $\psi \vee ((\neg\psi \wedge \varphi) \wedge X((\neg\psi \wedge \varphi)\ U\ \psi))$.

$\varphi\ W\ \psi$: instead of using a dedicated translation for the weak until operator, we simply rewrite it as $(\varphi\ U\ \psi) \vee G\ \varphi$.

Some explicit examples of automata generated using these rewritings, where states are labeled with the formula they are verifying 2.1 and 2.2:

On this example 2.2, we can see that the original formula has been rewritten in the initial state: the implies symbol has been desugared.

On that example 2.1, by reading the labels on the states and transitions on the downmost state, we can see that the formula $b\ U\ c$ has been rewritten as $(c \wedge \top) \vee (b \wedge \neg c \wedge X(b\ U\ c))$.

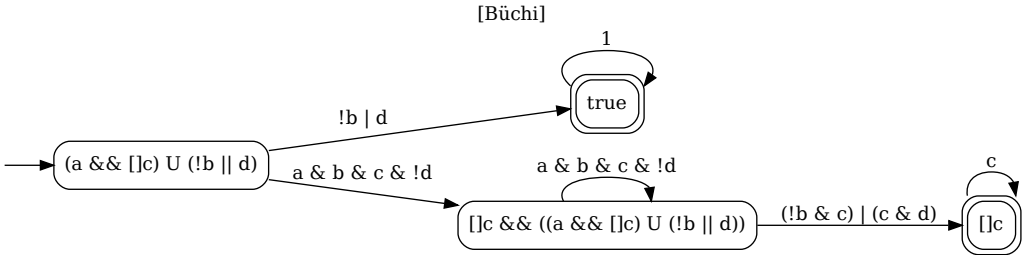



Figure 2.2: DFA for $(a \wedge G\ c)\ U\ (b \implies d)$

## 2.3 Formula graphs

To assist in the construction of the automaton resulting from a given formula $\varphi$, we introduce formula graphs (fgraphs for short). Formula graphs are a way to cache translated formulas and visualize the imbricated rewritings of formulas.

When a formula is encountered, it is first added to the formula graph as is. If it needs to be rewritten, we add its rewriting to the formula graph and specify to which formula the original is rewritten, and where to return on the rewritten formula.

Reading the fgraph of 2.3 from bottom to top, the first formula we encounter is on state 2, with $a\ U\ (b\ U\ c)$. Its successor is 4, which means it is rewritten in state 4.

Starting from state 4, we have two choices: either we verify $b\ U\ c$, or we have $a$ and need to verify $b\ U\ c$ on the next state.

We encounter the formula $X(a\ U\ (b\ U\ c))$ and add it to the fgraph.

Since we're already rewritten $a\ U\ (b\ U\ c)$, we can rewrite the other formula we've yet to rewrite: $b\ U\ c$.

For $b\ U\ c$, we either obtain $c$ directly or verify $b$ and need to verify the whole formula on the next state.

We encounter $X(b\ U\ c)$ and add it to the formula graph.

We encounter $b\ U\ c$, but since we already translated it, we end the rewritings here.

Now, for the fgraph 2.4 generated from the formula $\varphi = (a \wedge G\ c)\ U\ (b \implies d)$:

The formula is first rewritten as either obtaining $\neg b \vee d$ and accepting on the next state or obtaining $a \wedge G\ c \wedge \neg(\neg b \vee d) \wedge X(\varphi)$ which is equivalent to $a \wedge c \wedge \neg(\neg b \vee d) \wedge X(G\ c \wedge \varphi)$.

From then on, all states must verify $G\ c$. The rest of the rewriting is similar to the previous

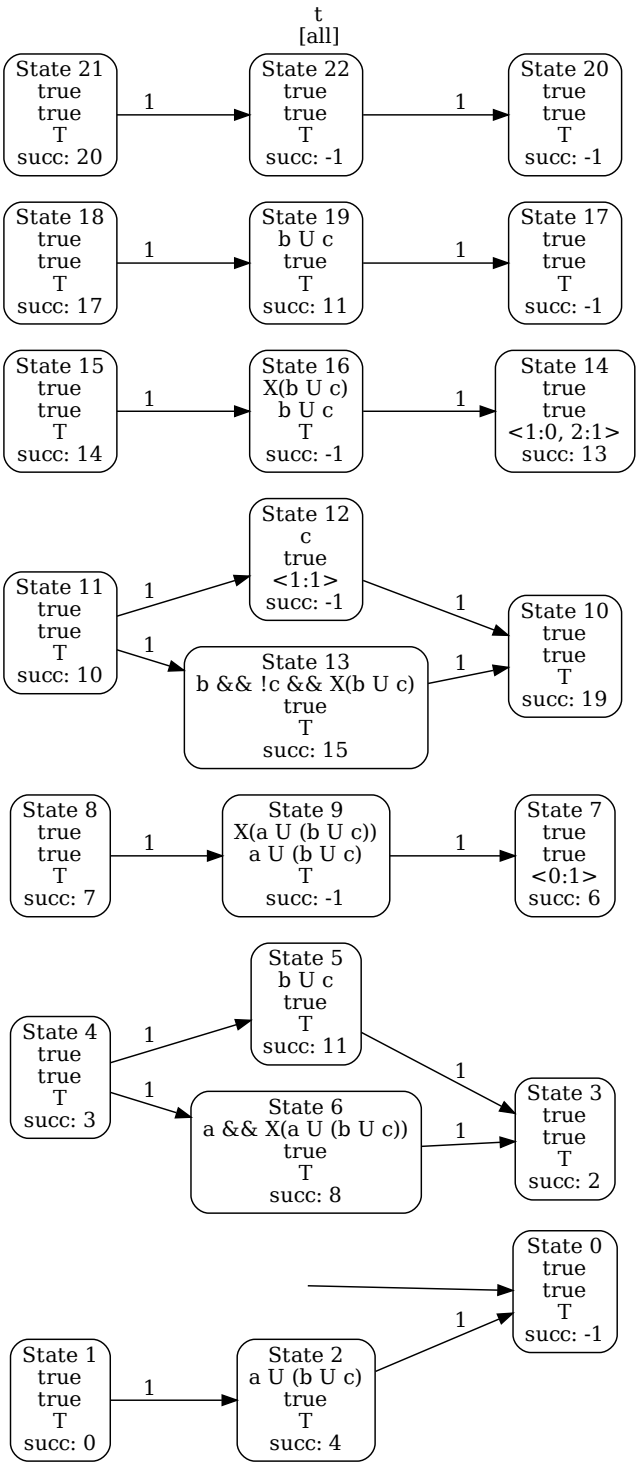



Figure 2.3: Formula graph for $a\ U\ (b\ U\ c)$

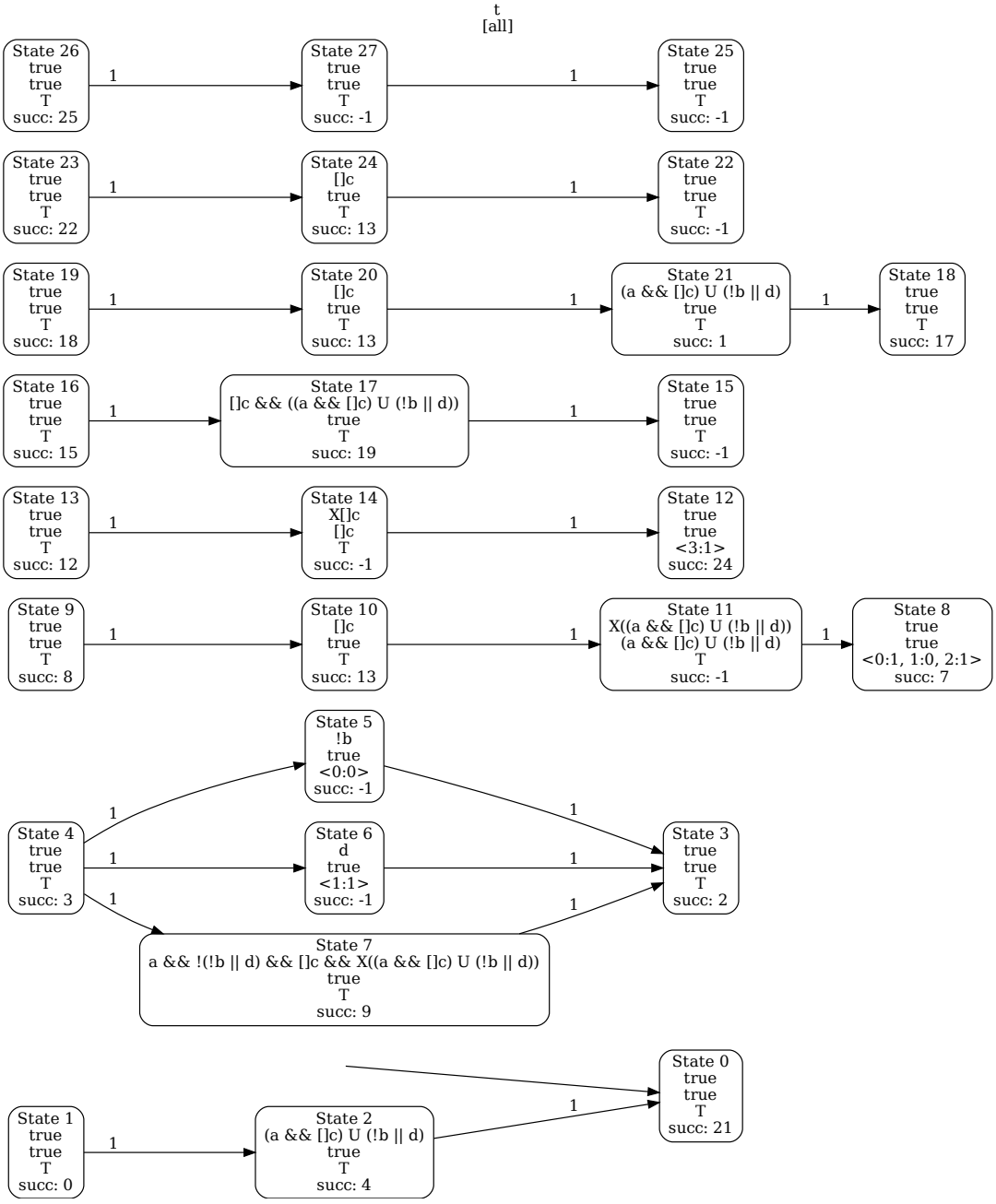



Figure 2.4: Formula graph for $a\ U\ (b\ U\ c)$

state with this added requirement.

## 2.4 Linear forms

In order to assist in translation and rewriting, we use linear forms to express all the outgoing transitions of a state. Transitions are represented by a pair of bdd and formula representing the condition of the transition and the formula it leads to. The linear form of a formula simply is the vector containing all outgoing transitions from this state.

From 2.1, we can easily reconstruct the initial state's linear form: we can reach $a\ U\ (b\ U\ c)$ by verifying $a$, $T$ by verifying $c$ and $b\ U\ c$ by verifying $b \wedge \neg c$. The resulting linear form is $[(a, a\ U\ (b\ U\ c)), (c, T), (b \wedge \neg c, b\ U\ c)]$.

### 2.4.1 Determinization and minimization of linear forms

The previously mentioned rewriting rules for LTLf formulas may cause linear forms to be nondeterministic and contain useless transitions. In order to resolve this issue, we introduce two algorithms for determinization and transition merging.

#### Determinization

A linear form is said to be nondeterministic if there exists two pairs $((c_1, \varphi_1), (c_2, \varphi_2))$ such that $c_1 \wedge c_2 \neq \bot$.

As a result, the process of determinization consists of splitting or partitioning all transitions such that none of the conditions overlap.

In order to avoid the complexity of going through each of the minterms, we all all transitions to an implication graph whose leaves are all disjoint and not necessarily minterms.

In the following algorithm, we consider the implication graph representing the partition of the BDDs to have been constructed:

For instance, let us study what the nondeterministic and deterministic automata created from $\varphi = G\ (p \implies X\ q)$ would yield:
A visual representation of the determinization process from 2.7 to 2.8.

The complexity of determinization is at worst quadratic in the number of transitions of the linear form. The number of translated formulas however doesn't change, because every subformula was going to be processed already and added to the formula graph.

**Data:** $lf$, the input linear form, $s$ its size, $g$ the implication graph, $m$ the map from the leaves of the graph (BDDs) to sets of formulas
**Result:** $lf_{det}$, the determinized linear form
**foreach** $(c, f) \in lf$ **do**
  $c_{set} \leftarrow$ leaves_of$(g, c)$ ; `/* The set of leaves forming c */`
  **foreach** $c_l \in c_{set}$ **do**
    append$(m[c_l], f)$ ; `/* f can be reached by c_l */`
  **end**
**end**
**foreach** $c \in$ *leaves*$(g)$ **do**
  $f_{det} \leftarrow \bigvee_{f \in m[c]} f$ ; `/* formula disjunction */`
  append$(lf, (c, f_{det}))$;
**end**

**Algorithm 1:** The determinization algorithm

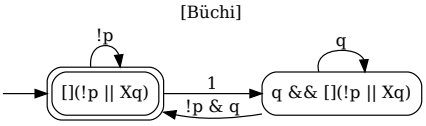



Figure 2.5: NFA constructed for $G\ (p \implies X\ q)$

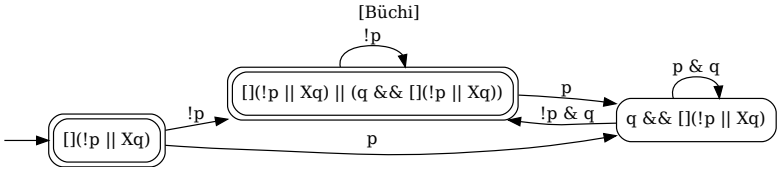



Figure 2.6: DFA constructed for $G\ (p \implies X\ q)$

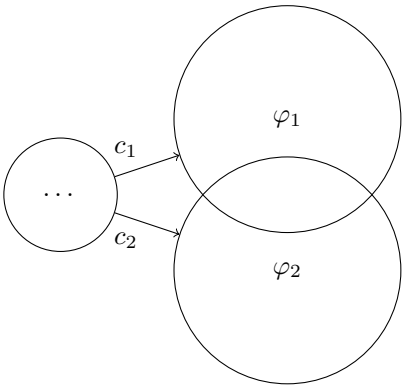



Figure 2.7: Original linear form

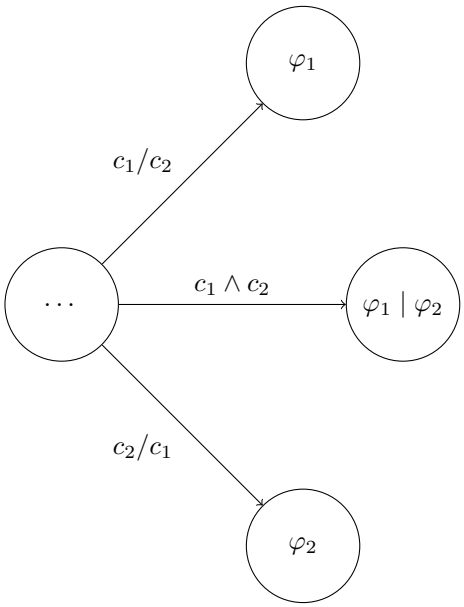



Figure 2.8: Determinized linear form

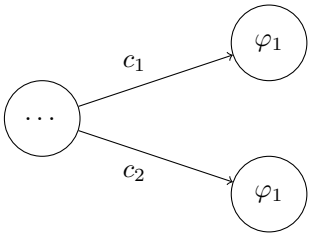



Figure 2.9: Original linear form

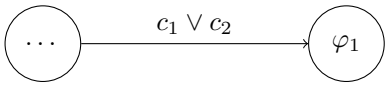



Figure 2.10: Merged linear form

### Transition merging

When the linear form contains transitions which lead to the same formula, we can simply combine all conditions using the Boolean or operator.

The resulting algorithm is:

**Data:** $lf$, the input linear form and $s$ its size
**Result:** $lf$, modified in place
$i \leftarrow 0$;
**while** $i < s$ **do**
  $j \leftarrow i + 1$;
  **while** $j < s$ **do**
    $(c_i, f_i) \leftarrow lf[i]$;
    $(c_j, f_j) \leftarrow lf[j]$;
    **if** $f_i = f_j$ **then**
      $c_i \leftarrow c_i \vee c_j$;
      swap$(lf[j], lf[s-1])$;
      pop$(lf)$;
      $s \leftarrow s - 1$;
    **end**
    $j \leftarrow j + +1$;
  **end**
  $i \leftarrow i + 1$;
**end**

**Algorithm 2:** The transition merging algorithm

Figures 2.9 and 2.10 present the idea of the transition merging algorithm.

The complexity of merging transitions is at worst quadratic in the number of transitions of the linear form, but merging all transitions into one imply less formulas to be translated in the next iteration of the translation algorithm.

# Chapter 3

# Model checking

Model checking is a formal verification technique used to systematically explore the reachable states of a system to ensure that the induced runs never violate the specification. In this approach, a model of the system, represented as a symbolic automaton, is checked against LTLf specifications to verify correctness.

Model checking with LTLf generally involves translating temporal logic specifications into (deterministic) finite automata, enabling the verification of finite traces to ensure that the system conforms to the given specification. Model checking is a key method in verifying system behavior, ensuring reliability and correct behavior.

## 3.1 Symbolic automata

In contrast to semi-symbolic automata, fully symbolic automata represent the entire automaton associated with a LTL formula with a single Boolean function represented as a BDD. The initial state is itself a Boolean function, and in order to go through transitions of the automata, we simply compute the intersection of the automaton's transition system $T$ with the current state $C$. We then obtain a Boolean function of the form $C \wedge T \wedge C'$ where $C'$ corresponds to the successor of the current state $C$.

It is important to note that states in the symbolic automaton are not equivalent to the states of the semi-symbolic automaton, but are a disjunction of the possible states the system can be in.

To determine whether states are terminating or not, we can split them using the terminating signal **termsig** which is a special Boolean formula indicating the termination of a run.

To continue exploring the automaton, we must compute the existential quantification of $T \wedge C \wedge \neg$**termsig** and map the result of the computation from primed states to regular states.

While this model is less explicit than the semi-symbolic automata seen prior, they make traversing the automaton more efficient, removing some constraints of nondeterministic finite au-

tomata.

## 3.2 Model checking on the fly

On the fly LTLf model checking offers multiple advantages or optimizations over classical model checking. The first compelling advantage is the possibility to detect the possibility of rejecting runs faster as the automata is incrementally constructed as it is explored. The incremental construction also benefits memory consumption and computational overhead, especially for runs which are rejected early on.

The compact representation of states and transitions using Boolean formulas manage complexity more efficiently. Combining on the fly construction with symbolic automata has the potential to accelerate the verification process of simple to complex systems.

The implementation is as follows:

Given a formula $\varphi$ representing a LTLf specification and a symbolic automaton corresponding to the system we want to verify, we would like to assure that there exists no possible run of the system which is accepted by $\neg\varphi$.

We perform a simultaneous depth first traversal of the symbolic automaton and the on the fly constructed automaton of the formula $\varphi$, starting from the pair $(\varphi, I)$ where $I$ corresponds to the initial state of the symbolic automaton.

For each pair of formula and symbolic state $(\psi, C)$, we start by computing $C'_{term}$ and $C'_{no\_term}$, respectively the terminating next primed state and non-terminating next primed state, as well as the linear form of the formula.

In the case of the terminating state, we check if there exists any transition in the linear form whose condition is compatible with the symbolic automaton's transition and which leads to a final state in the automaton constructed on the fly. If this is the case, then we have found a trace both accepted by the negation of the specification and by the system: the system violates the specification.

In the case of the non-terminating state, we need to keep exploring the automata. For each transition in the linear form, we constraint the transition in the symbolic automaton to the condition of the transition. If the transition is still possible, then we continue exploring the automata.

While performing the traversal, we need to check if we are processing a state we are already visiting, which could indicate undesirable loops in the controller. To do so, we keep a map of alive states to which we add states when they're currently being explored and remove upon completion of the traversal.

In order to avoid revisiting formulas we know do not lead to any final states, we keep track of dead states, which are states we have already explored and know have yielded no results for a given constraint.

Unfortunately, the algorithm has yet to be perfected and most important, implemented. As a consequence, no benchmarks could yet be made for model checking on the fly.

# Conclusion

In this technical report, we have studied the problems of synthesis and model checking algorithm given a LTLf specification. Our contributions include the translation algorithm from LTLf to NFA and DFA using the open source tool spot Duret-Lutz et al. (2022), as well as the groundwork for an on the fly model checking algorithm.

In the future, we would like to benchmark the already existing algorithms as well as implement the model checking algorithm to determine in which cases building the automaton on the fly benefits the verification.

## Acknowledgments

I would like to thank my supervisor Philipp Schlehuber-Caissier whose feedback and guidance were invaluable in writing this report. Furthermore, I would like to thank Quentin and Rania for assisting in the research at the LRE, as well as Sofia and Jude for their help in proofreading this report.

# Chapter 4

# Bibliography


Bacchus, F. and Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22. (page 9)

Camacho, A., Baier, J., Muise, C., and McIlraith, S. (2018). Finite ltl synthesis as planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 28(1):29–38. (page 10)

De Giacomo, G., De Masellis, R., and Montali, M. (2014). Reasoning on ltl on finite traces: Insensitivity to infiniteness. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1). (page 10)

De Giacomo, G. and Favorito, M. (2021). Compositional approach to translate ltlf/ldlf into deterministic finite automata. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):122–130. (page 10)

De Giacomo, G. and Vardi, M. Y. (2013). Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 854–860. AAAI Press. (pages 8 and 9)

De Giacomo, G. and Vardi, M. Y. (2015). Synthesis for ltl and ldl on finite traces. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 1558–1564. AAAI Press. (page 10)

Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A. G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., and Lauko, H. (2022). From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187. Springer. (pages 1 and 24)

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. (page 5)

Vardi, M. Y. (1996). An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher Order Workshop Conference on Logics for Concurrency : Structure versus Automata : Structure versus Automata*, page 238–266, Berlin, Heidelberg. Springer-Verlag. (page 11)

Zhu, S., Tabajara, L. M., Li, J., Pu, G., and Vardi, M. Y. (2017). Symbolic ltlf synthesis. (page 10)