Constructive Completeness Proofs and Model Searching in Temporal Logics

MPRI internship of Anatole Leterrier, supervised by Sam van Gool at IRIF

20–08–2022

General context

Temporal logics are at the heart of the domain of program verification. When reasoning about the behavior of a program, not only do we want to know which properties are true at the present, but we also need to consider what might happen at some time in the future. Along with the usual propositional language, these logics include temporal operators like 'next', 'future', or others characterizing the existence of a branch in a tree where a property is true infinitely often. As with any logic, we would like to prove that the chosen syntactical axiomatization generates the set of all semantically valid properties. Such proofs often use techniques from Stone duality, Büchi automata and tableaux, as I also do in this report.

Problem studied

The completeness of Linear Temporal Logic (LTL), the most straightforward of those logics, is a very well known result. It provides a theoretical foundation for algorithms solving the satisfiability problem, which asks to decide whether a formula has a semantical model. Other, more complex temporal logics exist, such as CTL (Computational Tree Logic). A recent proof of completeness of an extension of CTL with local fairness constraints [2] uses Stone duality as a tool in order to reason about axioms as properties of relations over Kripke structures.

While satisfiability of a formula can be decided for those logics, computing an actual model is another problem. We would like to find an algorithm which contructs one such model, if possible. Apart from the practical benefits of such an algorithm, this research provides global information about how temporal logics work. It is not always obvious how to extract a model from existing algorithms, as some of the proofs we mentioned are not constructive: they make use of ultrafilters in infinite algebras, which are sets of formulas, and generally cannot be characterized by a finite subset.

Contributions

This report, along with the accompanying code repository, presents the following contributions:

- In section 2, I compare the several ways in which LTL completeness and the SAT problem are handled in the literature. SAT has multiple approaches using Büchi automata which recognize particular LTL formulas, but tableaux-oriented methods have also been striving. I clarify the link between states of the automata associated with the formula, and atoms in a particular algebra generated by the formula, and I explain the theoretical principle of tableaux, relying on detailed examples.

- In section 3, I present an overwiew of a prototype OCaml implementation, which I engineered for Reynolds' tableaux algorithm for solving SAT in LTL [6], a most recent work in this vein. It can be found in this repository:

https://github.com/anatcaramba/LTL-SAT-Solver-by-Reynolds-Tableaux

Along with the binary answer to the SAT problem, it shows how the search for a model unravels, and provides one when it exists.

- In section 4, I provide an in-depth study of 'CTL^f-algebras' in order to characterize CTL^f-formulas by atoms. This is a first step into making the proof in [2] constructive.

Arguments supporting their validity

- Our characterization of the states of a Büchi automaton associated with a LTL formula in terms of atoms in an algebra generated by the formula has proven to give new insight on the connection between existing approaches to LTL satisfiability and Stone duality. It paves the way for future work on more complex logics.

- The OCaml implementation has proven to be conclusive on a comprehensive set of test cases, thanks to termination of Reynolds' tableaux algorithm. We provide a graph of complexity in time, for a worst-case analysis. This prototype shows that an approach based on functional programming makes sense, and those elements, combined with object-oriented tools, may be a source of improvement for the existing implementations in C++.

Summary and future work

This internship has revolved around deeply understanding the different tools used to evaluate whether formulas in temporal logics are satisfiable. A prototype OCaml SAT solver was programmed, not for performance purposes, but to understand how an algorithm can deal with *eventualities*, and actually compute a finite representation of an infinite model for formulas which involve them. In future work, one could try to adapt these methods to more complex temporal logics, such as CTL, CTL^f (for which we give first steps in section 4), or even temporal logics for trees with counters.

1 Introduction

This section contains a few preliminaries, to make the reader more familiar with the content at discuss in this report. Beyond the definitions given in the section, we will use a few basic notions of finite automata, Boolean algebra and logics, which can be found respectively in the course: https://www.irif.fr/ jep/PDF/MPRI/MPRI.pdf Section III.3, in [1, Section 1.2], and in [5, Section 3].

1.1 LTL (Linear Temporal Logic)

In the most general setting, logics are sets of structural rules, which are intended to formalize an ensemble of true properties of some system. Although the origins of logics can be dated back to Aristotle's syllogisms, they found new life with the advent of computer science, as a tool for verification of systems, and automation of proofs. It is hard for a computer to prove a theorem on its own, using the human way of reasoning; however, bringing a theory down to a finite set of rules, the search for proofs falls into the understanding of computer systems. Throughout this report, we will speak of *syntax* when considering the formal system of the logic: axioms and inference rules. On the other hand, *semantics* define the actual meaning of formulas; they tell us whether we should consider a formula to be satisfied in some system we chose.

The simplest logic is the classical propositional calculus, defined for instance in [5, Section 3]. The set of formulas of propositional logic over a set of Boolean variables $\{p_1, \ldots, p_n\}$ can be restated as the *Free Boolean Algebra* $(A, \vee, \neg, \bot)$ over this set [1, Section 4.1], where the classical, well-known implications and equivalences over Boolean algebras hold. From now on, we will prefer condensing syntax of logics into algebraic terms, as a compact restatement of the set of rules. This will allow us to avoid long lists of inference rules, focusing instead on the structure of our logics.

Temporal logics were first introduced by A. Pnueli in 1977 in [4], and can be viewed as an enrichment of propositional calculus with time constraints. We begin with a simple instance of such a logic, that we focus on in sections 2 and 3 of this report.

Definition 1. The unary fragment of *Linear Temporal Logic* (LTL) consists of the language of propositional logic, along with operators defined on every formula:

- $\mathbf{X}$ is the *Next* operator. Syntactically, it is defined as a Boolean endomorphism over the Boolean algebra $(A, \vee, \neg, \bot)$. Semantically, if φ is a formula of LTL, $\mathbf{X}\varphi$ is satisfied in a state if φ is satisfied in the next state. We will make precise below what this 'next state' is in the context of our work.

- $\mathbf{F}$ is the *Future* operator, which also stands for 'finally'. For every formula φ in LTL, $\mathbf{F}\varphi$ is defined as the least pre-fixpoint of the function $x \mapsto \varphi \vee \mathbf{XF}x$. Semantically, $\mathbf{F}\varphi$ is satisfied whenever there exists a state in the future (or in the present) where φ holds. Algrebraically, $\mathbf{F}$ is a *unary operator*, which also is *join-preserving*: $\mathbf{F}(\varphi \vee \psi) = \mathbf{F}\varphi \vee \mathbf{F}\psi$.

- $\mathbf{G}$ *(Globally)* is a syntactical shortcut for $\neg\mathbf{F}\neg$, which means that $\mathbf{G}\varphi$ is true exactly when φ holds forever from the current state on. By symmetry, it is a meet-preserving, unary operator: $\mathbf{G}(\varphi \wedge \psi) = \mathbf{G}\varphi \wedge \mathbf{G}\psi$.

Note that the definition of LTL usually includes operator $\mathbf{U}$ (*until*), which we will not discuss here. Another remark: while it holds for every φ that $\varphi \vee \mathbf{XF}\varphi \leq \mathbf{F}\varphi$, we actually get an equality for this last inequality. Indeed, $\mathbf{X}(\mathbf{XF}\varphi\vee\varphi) = \mathbf{X}\varphi\vee\mathbf{XXF}\varphi$, which is below $\mathbf{XF}\varphi\vee\mathbf{XF}\varphi$, by definition of $\mathbf{F}$ and by monotonicity of $\mathbf{X}$. This is itself below $\mathbf{XF}\varphi\vee\varphi$. Now, by minimality in the definition of $\mathbf{F}$, it holds that $\mathbf{F}\varphi \leq \mathbf{F}(\mathbf{XF}\varphi\vee\varphi) \leq \mathbf{XF}\varphi\vee\varphi$, which proves that the first inequality exactly is an equality.

Now, we make precise the semantical meaning of 'next' and 'future' operators.

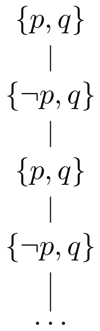
Definition 2. An *infinite word* over some set S is a sequence $(s_n)_{n\in\mathbf{N}}$ of elements (or *nodes*) in S. Finally, for a finite set $\bar{p}$ of variables, a *$\bar{p}$-coloring* is a function $\sigma : S \to \mathcal{P}(\bar{p})$.

We can define the forcing relation $\models$ between nodes in S and formulas φ with variables in $\bar{p}$, by induction on φ:

- $s \not\models \bot$;
- $s \models p$ if $p \in \sigma(s)$;
- $s \models \neg\varphi$ if not $s \models \varphi$;
- $s \models \varphi \vee \psi$ if $s \models \varphi$ or if $s \models \psi$;
- $s \models \mathbf{X}\varphi$ if $s' \models \varphi$, where s' denotes the unique successor of s;
- $s \models \mathbf{F}\varphi$ if there exists $n \geq 0$, such that in the unique path $s_0, s_1 \dots s_n$ ($s_0 = s$), $s_n \models \varphi$;

Also, note that it follows from the definitions that $s \models \mathbf{G}\varphi$ if and only if in the unique path $s = s_0, s_1 \dots s_n$, we have $s_n \models \varphi$ for every $n \geq 0$.

Example 3. We now give an instance of an infinite word, colored over the two-variable set $\{p, q\}$. The variable q is always set to 'true', while the variable p alternates between 'true' and 'false' at each state. This is a fairly common example, which we use as a test case in our implementation in section 3.



Examples of formulas which are satisfied by this word are $p\wedge q$, $\mathbf{X}\neg p$, or the more complex $p \wedge \mathbf{G}(q) \wedge \mathbf{G}((p \wedge \mathbf{X}\neg p) \vee (\neg p \wedge \mathbf{X}p))$, which describes exactly the nature of this model, with q always set to true and with p alternating. However, $\mathbf{F}(\neg p \wedge \mathbf{X}\neg p)$ is not satisfied here, as we will never reach two consecutive states where p is set to 'false'.

1.2 Fair CTL (Computational Tree Logic)

CTL is a very common temporal logic, operating over infinite trees instead of infinite words. It consists of operators such as AX ('all successors verify. . . ') and EG ('there exists a branch where globally . . . holds'). However, in [2], a further variant named CTL^f was introduced, allowing to add local fairness constraints on branches. Let us define precisely what it means.

Syntax of fair CTL

Definition 4. Let $\bar{p} = \{p_1, \dots, p_n\}$ be a finite set of propositional variables. We define inductively a *CTL^f-formula* φ to be of the shape:

$$\varphi = \bot \mid p_i \mid \neg\varphi \mid \varphi \vee \psi \mid \Diamond\varphi \mid EU(\varphi, \psi) \mid EG(\varphi, \psi)$$

For convenience, we then also define the De Morgan duals of our binary operations, respectively $\wedge$ for $\vee$; AR for EU; AF for EG. For the unary operator, we set $\Box\varphi = \neg\Diamond(\neg\varphi)$; Finally, we set $\top$ to be the negation of $\bot$, naturally.

Definition 5. A *CTL^f-algebra*, is a tuple $\mathbb{A} = (A, \bot, \neg, \vee, \Diamond, EU, EG)$ which verifies each one of the following axioms:

1. Boolean algebra axioms for $\vee, \neg, \bot$;
2. Unary operator $\Diamond$ preserves finite joins, including the empty join $\bot$;
3. $\Diamond\top = \top$;
4. Binary operators EU and EG satisfy the following *fixpoint axioms* for all a, b, c:

   - $a \vee (b \wedge \Diamond EU(a, b)) \leq EU(a, b)$
   - $a \vee (b \wedge \Diamond c) \leq c \implies EU(a, b) \leq c$
   - $EG(a, b) \leq a \wedge \Diamond EU(b \wedge EG(a, b), a)$
   - $c \leq a \wedge \Diamond EU(b \wedge c, a) \implies c \leq EG(a, b)$

   In other words, $EU(a, b)$ is the *least pre-fixpoint* of the function $x \mapsto a \vee (b \wedge \Diamond x)$. And $EG(a, b)$ is the *greatest post-fixpoint* of the function $x \mapsto a \wedge \Diamond EU(b \wedge x, a)$.

Importantly, we remark that for all a, b, $AR(a, b)$ is the greatest post-fixpoint of $x \mapsto a \wedge (b \vee \Box x)$, while $AF(a, b)$ is the least pre-fixpoint of $x \mapsto a \vee \Box AR(b \vee c, a)$.

Once again, the algebraic structure we define corresponds to what we expect of a syntax. That is, there are *equations* (equalities) which correspond to axioms of a deduction system, like:

$$\frac{}{\top \vdash \Diamond\top}$$

But there also are *quasi-equations* (implications) which translate to deduction rules:

$$\frac{c \vdash a \wedge \Diamond EU(b \wedge c, a)}{c \vdash EG(a, b)}$$

However, we never use the rules properly speaking, but we define a structure which embodies those properties and use operations on it. This is sometimes called *algebraic semantics.*

Also, note that the axioms on EU and EG show that CTL^f is a fragment of the modal *μ-calculus*, see [7, Section 2]. For example, for every a, b, $EU(a, b)$ can be defined as $\mu x.(a \vee (b \wedge \Diamond x))$, the least fixpoint of this monotone function (it is enough to say the least *pre-fixpoint*, for reasons similar to those we gave for LTL. As for EG, it is a greatest *(post-)* fixpoint, so it could have been defined with a ν operator, where $\nu f = \neg\mu(\neg f)$. More precisely:

$$EG(a, b) = \nu y.(a \wedge \Diamond\mu x.((b \wedge y) \vee (a \wedge \Diamond x))).$$

This makes apparent a nesting of a μ operator inside a ν operator. This is what makes the logic hard to analyze, compared to LTL which only contains one fixpoint operator, without alternations of μ and ν.

Definition 6. For any finite set of propositional variables $\bar{p}$ and any CTL^f-algebra $\mathbb{A}$, we can define a *valuation* $V : \bar{p} \to A$. Then any CTL^f-formula φ with variables in $\bar{p}$ can be interpreted as a term in $\mathbb{A}$ (the *interpretation* of φ under V, $\varphi^{\mathbb{A}}(V)$).

Then, we can say the equality $\varphi(\bar{p}) = \psi(\bar{p})$ is *valid* if it is true under every valuation to every A.

φ and ψ are said to be *equivalent* if the equation $\varphi = \psi$ is valid. φ is called a *tautology* if it is equivalent to $\top$, and is said to be *consistent* if it is not equivalent to $\bot$.

Finally, we say φ *entails* ψ if $\neg\varphi \vee \psi$ is a tautology, which we will denote $\varphi \vdash \psi$.

Semantics of fair CTL

Definition 7. We first define the notion of *transition system*, i.e. a pair (S, R), where S is a set, and R a binary relation on S. Then, an *R-path* is a (possibly infinite) sequence of nodes in S such that $s_i R s_{i+1}$ for all i. Finally, for a finite set $\bar{p}$ of variables, define a *$\bar{p}$-coloring* $\sigma : S \to \mathcal{P}(\bar{p})$ as above.

We can define the forcing relation $\models$ between nodes in S and formulas φ with variables in $\bar{p}$, by induction on φ:

- propositional cases are defined identically as in LTL;
- $s \models \Diamond\varphi$ if there exists s' such that sRs' and $s' \models \varphi$;
- $s \models EU(\varphi, \psi)$ if there exists $n \geq 0$ and a path $s_0 R s_1 R \ldots R s_n$ $(s_0 = s)$ such that $s_k \models \psi$ for every $k < n$ and $s_n \models \varphi$;
- $s \models EG(\varphi, \psi)$ if there exists an infinite path $s_0 R s_1 R \ldots$ $(s_0 = s)$ such that $s_k \models \varphi$ for every k and $s_j \models \psi$ infinitely often on the path.

Note that as a consequence, $s \models \Box\varphi$ if and only if for every successor s' of s by R, $s' \models \varphi$ holds. Also, $s \models AR(\varphi, \psi)$ if and only if for every $n \geq 0$ and every path $s = s_0 R s_1 R \ldots R s_n$, either $s_k \models \psi$ for some $k < n$, or $s_n \models \varphi$.

Similarly, $s \models AF(\varphi, \psi)$ if and only if for all infinite paths $(s)_k$ starting from s_0, if $s_j \not\models \psi$ infinitely often, then $s_k \models \varphi$ for some k.

Also, as a convention, we will require the transition systems to be *serial*, i.e. that every node has a successor. Syntactically, this is represented by the axiom $\Diamond\top = \top$.

2 Two approaches to completeness of LTL

Once syntax and semantics of a particular logic are defined, the goal is to prove that the formulas they say to be true are the same, in order for a computer system to use it and be trusted. The first part is to prove that the syntactical (or logical) system is *sound* with regard to the semantics, i.e. that every formula which can derived to be a *tautology* (equivalent to $\top$) in the system is actually *valid* (true under every valuation) semantically. Soundness is usually proved by induction, examining the last rule of inference used to derive the tautology.

Completeness is the reciprocal implication, and proving it can be much harder. In the cases which are of interest to us, completeness is proved by solving the *SAT problem*, which is the question of whether a formula has a model (is satisfiable by a valuation). Indeed, the following are equivalent:

$$\forall\varphi(\varphi \text{ is valid } \Rightarrow \varphi \equiv \top)$$
$$\forall\varphi(\text{not}(\varphi \equiv \top) \Rightarrow \varphi \text{ is not valid})$$
$$\forall\varphi(\neg\varphi \not\equiv \bot \Rightarrow \neg\varphi \text{ is satisfiable})$$
$$\forall\psi(\psi \not\equiv \bot \Rightarrow \psi \text{ is satisfiable})$$

Then it is sufficient to show that every *consistent* formula (not equivalent to $\bot$) is satisfiable in the system considered. Sometimes, proofs will first establish an equivalence between satisfiability over infinite words and satisfiability in some system designed to be easy to manipulate. These systems may be automata, over which usual operations (conjunction, complementation, etc.) can be performed; or they may use a tableau decision procedure, which is algorithm-friendly.

This is why this section will focus on the different ways the SAT problem is handled in LTL. In the literature, we distinguish between two major families of approaches: automata- and tableaux- based.

2.1 Automata for LTL

Computing finite automata is a natural way of tackling the satisfiability problem in logics. By constructing a mathematical object which accepts only a chosen set of models, we can perform usual transformations such as reduction, giving a comprehensive understanding of the semantical meaning of a formula. We remind that basic information can be found in this course about automata: https://www.irif.fr/ jep/PDF/MPRI/MPRI.pdf

The approch we will describe comes from F. Laroussinie's MPRI course [10, pages 22-36]; variants can be found in [11, Section 3] and [12, Section 1.3]. Also take a look at this implementation in C (http://www.lsv.fr/ gastin/ltl2ba/index.php), computing accepting Büchi automata for LTL formulas (which is presented in [13, Pages 53-65]).

This particular approach uses *generalized Büchi automata*, which are tuples $(Q, q_0, \to, \mathcal{F})$, where $\mathcal{F}$ is not a set of final states but a *set of sets* $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ of final states. A path will be accepted if it goes infinitely often in each of the $\mathcal{F}_i$. The idea is that every state of the automata represents a set of formulas which we want to be true.

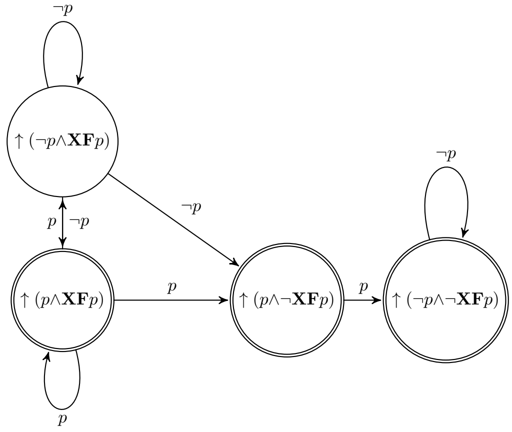
Here is how the author constructs an automata $\mathcal{A}_\varphi$ recognizing an LTL formula φ with variables in $\bar{p}$ (this report only deals with unary LTL). First of all, formulas in the states' labels are among the finite set $\mathcal{S}_\varphi$, consisting of subformulas of φ along with their negations, and $\mathbf{XF}\psi$ for every eventuality $\mathbf{F}\psi$. Then, available states must be labeled by states of formulas which are 'coherent, maximal and conform to Unary LTL semantics'. We restate this with our own words, saying that *labels of states must be ultrafilters in the Unary-LTL-algebra $\mathcal{A}_\varphi$ generated by $\mathcal{S}_\varphi$*, see [1, Section 4.1]. Concretely, those sets must not contain contradictory formulas, and if a set contains some subformula ψ, it must also contain its universal consequences (it is *upwards-closed*).

Then, we set a transition between states q and q' if and only if:

$$\begin{cases} \text{for every } \mathbf{X}\psi \text{ in } q, \ \psi \text{ is in } q' \\ \text{for every } \mathbf{F}\psi \text{ in } q, \ \text{either } \psi \text{ is in } q \text{ or } \mathbf{F}\psi \text{ is in } q' \end{cases}$$

And such a transition has for label in $\mathcal{P}(\bar{p})$ the set of propositional variables in q. Concretely, this means that transitioning between two states in the automata moves forward of one unit in time, and the label of the transition is none other than the chosen valuation of the corresponding node in an infinite model. Finally, initial states are those whose label contains φ, and a set of final states $\mathcal{F}_\psi = \{q \in Q \mid \psi \in q \text{ or } \mathbf{F}\psi \notin q\}$ is defined for every subformula $\mathbf{F}\psi$ in $\mathcal{S}_\varphi$, to ensure that eventualities are either satisfied, or unsatisfied at one point in time.

Example 8. Let us examine what the automaton looks like for a formula such as $\mathbf{F}p$. Here, we actually compute the automata which is associated with the (finite) algebra $\mathcal{A}_{\mathbf{F}p}$. As we explained earlier, every state corresponds to an ultrafilter of this algebra. We identified each state with the *atom* characterizing this ultrafilter [1, Section 1.2], i.e. the only element a in $\mathcal{A}_{\mathbf{F}p}$ such that $a > \bot$ and a is below every element in the ultrafilter.



$\neg p$
$\uparrow(\neg p \wedge \mathbf{XF}p)$
$\neg p$
$\neg p$
p
$\neg p$
$\uparrow(p \wedge \mathbf{XF}p)$
p
$\uparrow(p \wedge \neg\mathbf{XF}p)$
p
$\uparrow(\neg p \wedge \neg\mathbf{XF}p)$
p

Now, every state but the one located at the far right is initial, as $\mathbf{F}p$ is a logical consequence of all of the other three atoms. What is interesting in this approach is that the automaton is not minimal. Instead it lets us know about the deep structure of the algebra, and what every states stands for in $\mathcal{A}_{\mathbf{F}p}$. During this internship, we always tried to express syntax in algebraic terms, more particularly atoms and ultrafilters, and these automata allowed exactly that.

2.2 Tableaux for LTL

The theoretical information provided by automata is not always what we favor. If we lean towards a more practical approach, and if we want to compute a model for a formula from scratch, tableaux are closer to our goal. Tableaux are deicision procedures which unravel the subformulas that must be true if the initial formula is, branching in case of a disjunction, getting rid of impossible paths, and finally finding a model if there is one. Reynolds' tableaux algorithm ([6], 2016) for LTL, unlike methods based around automata, was conceived with implementation in mind. In the abstract, the author claims that his tree-style tableau is simple to understand, which we agree on: rules are designed in a straightforward way, and the visual representation of tableaux as trees enhances understanding.

The greatest feature of this paper is to use trees, instead of the more general graphs used by classic tableaux (see the procedure by Wolper [14], which was implemented and improved by Goranko [15], among others), which allows to make the most benefit out of recursion, not having to create massive data structures. We will briefly describe the algorithm, before discussing more precisely the details of our implementation in OCaml. Note that we still restrict the language to unary LTL in this section, that is, LTL without 'Until'.

We write the algorithm in pseudo-code, then explain the most important parts of it (a general understanding of tableaux is provided in subsection 2.2).

```
Input:  Unary-LTL formula $\varphi$
Select any order of exploration of the leaves
Set S $\leftarrow$ $\{\varphi\}$
Tree T $\leftarrow$ Root labeled S
While (There is at least one uncrossed leaf):
      S $\leftarrow$ Next uncrossed leaf in the order
      If S is empty:
           Return True
      Else if S contains $\bot$ :
           Cross leaf
           S$\leftarrow$next uncrossed leaf if there is one
      Else if S contains two contradictory formulas $\psi$ and $\neg\psi$:
           Cross leaf
           S$\leftarrow$next uncrossed leaf if there is one
      Else if a unary static rule applies:
           S'$\leftarrow$S updated along rule
           Add leaf labeled S' below S
      Else if a binary static rule applies:
           S1$\leftarrow$S updated along rule case 1
           S1$\leftarrow$S updated along rule case 2
           Add two leaves labeled S1, S2 below S
      Else if LOOP Rule applies:
           Return True
      Else if PRUNE Rule applies:
           S$\leftarrow$next uncrossed leaf if there is one
      Else if PRUNE$_0$ Rule applies:
           S$\leftarrow$next uncrossed leaf if there is one
      Else:
           S'$\leftarrow$S updated along TRANSITION Rule
           Add leaf labeled S' below S
Return False
```

The tree is first set to a single root labeled by φ. While there still are uncrossed leaves, the algorithm keeps exploring possible models. Without considering the 'dynamic rules' described below, a leaf is crossed if it contains $\bot$, or two contradictory formulas (which would therefore be impossible to satisfy at the same time).

Sets of formulas are unraveled until they contain the most basic information about the truth value they require for the variables. This is done along a set of *static rules*, which get rid of most operators ($\wedge$, $\vee$, $\mathbf{F}$, $\mathbf{G}$). Static rules are straightforward, so we will only specify a few:

($\wedge$ rule) if a set contains some formula of the form $\psi\wedge\chi$, it is replaced by both ψ and χ;

($\neg\neg$ rule) any $\neg\neg\psi$ in a set is replaced by ψ;

($\neg\mathbf{G}$ rule) if a set contains some $\neg\mathbf{G}\psi$, the tree forks into two cases, one with the formula replaced by $\neg\psi$, the other with $\mathbf{X}\neg\mathbf{G}\psi$.

... etc.

The most innovative part of the algorithm are the *dynamic rules*: LOOP, PRUNE, PRUNE$_0$, TRANSITION.

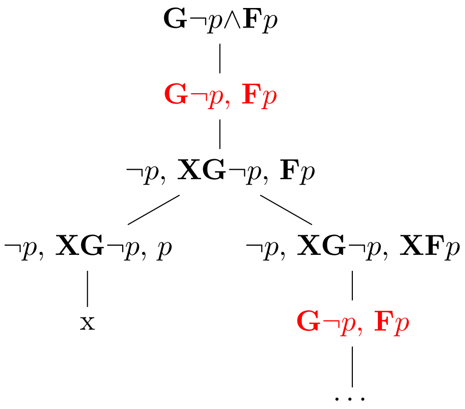
TRANSITION is the rule we apply to a leaf when nothing else is possible, meaning that we have simplified a set of formulas the most we could. It gets rid of literals (formulas of the form p or $\neg p$), which describe the valuation of the node in the model being constructed; then, it takes the remaining formulas in the set labeling the leaf, which all are of the form $\mathbf{X}\psi$ or $\neg\mathbf{X}\psi$, and removes the 'Next' operator, meaning that we have moved onto the next state in the model we are attempting to build.

LOOP, PRUNE (and PRUNE$_0$, which is a variant of PRUNE) are the rules which decide what happens when a branch becomes periodic without ever coming to a conclusion.

Let us talk specifically of PRUNE. We will state the rule as it is in Reynolds' paper ([6]), then explain its meaning; reasoning about the other rules is similar.

Definition 9. *(PRUNE rule)* Suppose that $u < v < w$ and each of u, v and w have the same label Γ, to which no static rule applies. Suppose also that for each $\mathbf{X}$-eventuality of the form $\mathbf{XF}\beta$ in Γ, if there is x with $\beta \in \Gamma_x$ and $v < x \leq w$ then there is y such that $\beta \in \Gamma_y$ and $u < y \leq v$. Then w is a crossed leaf.

Example 10. In the example below, we have highlighted that the tree periodically loops into the same state labeled by the set $\{\mathbf{G}\neg p, \mathbf{F}p\}$; however, the $\mathbf{X}$-eventuality $\mathbf{F}p$ is never fulfilled, as the left branch is discarded by the contradiction rule. The PRUNE rule makes sure that when formulas of the type $\mathbf{F}\psi$ are postponed indefinitely, the branch is discarded for not making any progress.



3 Reynolds' SAT Solver for LTL: an implementation

We recall that our implementation can be found here:

https://github.com/anatcaramba/LTL-SAT-Solver-by-Reynolds-Tableaux

More information is available there about prerequisites, execution, etc.

Why OCaml?

The choice of a functional programming language was guided by the close relationship between inductive definitions of the formulas and the pattern matching feature. Throughout the program, functions make use of matching, both to discriminate what we call 'eventuality formulas' from others, and to construct a printer for LTL formulas, by going into the possible imbrications of operators to construct a readable string.

Functional programming also comes with its setbacks. A quite unoptimized section of the program is loop checking (LOOP and PRUNE rules). We need to check a condition on elements in the list between two indexes. Unlike in an imperative language, accessing a specific element in a list is not immediate.

Another issue arises with the LOOP rule. As objects are immutable, we could not find an efficient way to compute a model when a branch is validated thanks to LOOP.

Note that the following implementation is a *prototype*, and as such it is experimental rather than optimized. In contrast, Leviathan [8] is an implementation of the algorithm in C++, which was developed by a group of researchers including Mark Reynolds; it is more complete than what is presented below, making use of parsers for inputs, telling users where to loop to find infinite models, etc. The tool can be downloaded on this webpage: https://github.com/Corralx/leviathan. In the same vein, 'BLACK' [9] is more recent and incorporates SAT solvers to resolve conflicts more quickly.

Overview of the implementation

Most of the program uses the newly defined `ltl` type, which is based on the inductive definition of unary LTL formulas we have given, and will allow to apply recursion nicely.

```
type ltl =
| Prop of char
| Top
| Bot
| Neg of ltl
| And of ltl * ltl
| Or of ltl * ltl
| F of ltl
| G of ltl
| X of ltl
```

Type `ltl_op` is an additional tool to characterize an LTL function by its main operator. The program contains a handful of useful functions such as `contains_contra`, `belongs_list` or `are_equal`, which often are tools made out of the standard OCaml librairies.

We have chosen to view a node of the tableau as a `ltl list list`. First of all, it makes perfect sense for a node to be represented as a `ltl list`, given that the algorithm unravels the subformulas which have to be satisfied if the formula has to be true. Then, we need to be able to keep trace of ancestors of every node, in case we need to check if rules LOOP, PRUNE, PRUNE$_0$ apply. However, since OCaml deals with immutable objects, it was not obvious to us how to create a global tree structure that we would update throughout the algorithm. Then the easiest way to deal with this was to pass the ancestors as parameters in our recursive process. This is why we treat every node as a `ltl list list`. A potential future improvement would be to actually create this tree structure.

The main function `sat` mostly follows the structure of Reynolds' algorithm [6]. In addition to that, it prints the unravelling of the tree, namely: specify which formulas are to be satisfied in each node, tell when there is branching, when a branch is discarded or validated and why. We tried to give the main information while keeping the output readable.

The most intricate functions in the program are those designed to test ‘dynamic’ termination rules: LOOP, PRUNE and PRUNE$_0$. For example, take the LOOP rule as defined in [6].

Definition 11. *(LOOP rule)* If a node v with label Γ_v has a proper ancestor (i.e. not itself) u with label Γ_u u such that no static rule applies to each label, $\Gamma_v \subseteq \Gamma_u$, and such that for each **X**-eventuality of the form $\mathbf{XF}\beta$ in Γ_u we have a node w such that $u < w \leq v$ and $\beta \in \Gamma_w$, then v can be a validated leaf.

Here is our function, which returns true if and only if the LOOP rule applies to the tableau.

```
let loop_applies (ll:ltl list list):bool=
    let i = poised_ancestors_contain ll in
        let current_list = List.hd ll in
            List.exists(fun k_v -> is_included (List.nth ll k_v)(current_list) &&
(* there is a proper poised ancestor v contained in current poised label and *)
                List.for_all(fun phi->List.exists (fun j->belongs_list phi
                    (List.nth ll j)) (range 1 (k_v+1) ))(f_X_ev (current_list)))i
(* every XF-ev in current label was satisfied after v *)
```

In what follows, a label is called *poised* if no static rule applies to it. The List.exists and List.for_all imbrication (lines 4 to 8), while not immediate to read, is a rewriting of the sentence in LOOP rule. However, the subtlety is that poised_ancestors_contain, given the current unraveling of the tableau ll, returns a list of *integers*, that is, the indices of the current node’s poised ancestors which contain the current node. This points to the set of nodes named u in the above definition. Then, tests such as List.exists access the actual poised label Γ_u with List.nth. This solution is unoptimized, as in OCaml this access takes linear time in the most general case (and not constant, as it would in C). A future improvement would be to track poised ancestors in a global structure, so that the program does not compute the same list of ancestors twice. At this point, it appears clearly that an efficient implementation of the algorithm would benefit from both functional- and object-oriented programming features.

As we hinted towards earlier in this section, we did not yet manage to give a precise model when satisfiability is proved thanks to the LOOP rule. The program simply tells to ‘loop from here’, but whether to repeat the last state indefinitely, or the last two, three or more states is left to the user. While the answer should not be too difficult in usual cases, for longer formulas the user could not do without advice. What prevents us to find the answer for now is, again, the nature of the loop_applies function. List.exists... returns true if an ancestor meets the right conditions, but not its index. So computation of models would also benefit from a global structure for the tree, keeping all necessary information in memory.

Performance

Among a battery of tests, we have created a sequence of formulas that exhibits the worst-case performance of the solver. It corresponds to the ‘exponential case’ test. We build a sequence of ‘worst-case formulas:

$$\begin{cases} \varphi_0 = \bot \\ \varphi_{n+1} = \varphi_n \vee \varphi_n \end{cases}$$

While idempotency of $\vee$ would make this case trivial, note that we could adapt by designing 2^n different leaves containing formulas equivalent (but not equal) to $\bot$. The spirit remains the same here. The formula in the test case only has rank 5, however we have tested up to 13 nestings. Since the procedure is PSPACE-complete [6, Section 9], we would expect an exponential time in function of the number of nestings in the worst case, which is what we seem to get.

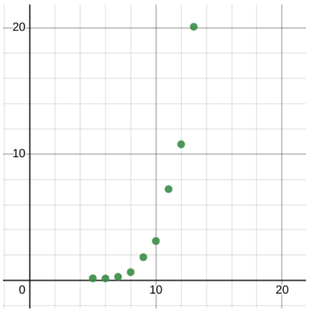


Figure 1: Evolution of the 'exponential case' test with the depth of nestings.

Number of nestings	5	6	7	8	9	10	11	12	13
Testing time (s)	0.17	0.16	0.29	0.66	1.84	3.12	7.23	10.78	20.08

This is normal, as adding a nesting doubles the number of leaves, all of which end up crossed (so the algorithm has to check every one of them).

Future improvements

- Upgrade the main function with global objects, keeping track of information invariant throughout the computation of a model. This would make the implementation faster, as well as compute precise models in the case they loop from a number of states. Also, the large number of recursions might saturate the space allowed by memory.

- Implement parsers to make the program more user-friendly. Rather than having to go through the top-level, the user should be able to execute the function directly in a terminal, with a tolerance for different syntaxes of formulas. The issue for now is being able to parse a string into an actual `ltl`-typed object.

- The chosen syntax makes it hard for longer formulas to be both read, and typed by hand. For user-comfort purposes, and to allow tests on formulas which could really reach the program's limit, we should be able to write scripts that automatically generate formulas.

- Perform more tests on interesting practical benchmarks.

4 Towards a constructive proof for completeness of fair CTL

Problem

The multitude of approaches that exist for proving LTL completeness inclines us to do adapt them to more complex temporal logics, and CTL^f in particular. Once again, our main concern is the actual computation of models. In other words, the question is whether we can find an answer to the SAT problem for CTL^f which is *constructive*; we would like to write an algorithm which tells us, for a CTL^f-formula φ, a transition system (S, R) and a finite set of variables $\bar{p}$, which $\bar{p}$-coloring $\sigma : S \to \mathcal{P}(\bar{p})$ (or even better, which set of colorings) of S satisfies φ.

As we can recall from section 1, both binary operators EG and EU could have been defined in terms of modal μ-calculus. In particular:

$$EG(a, b) = \nu y.(a \wedge \Diamond \mu x.((b \wedge y) \vee (a \wedge \Diamond x))).$$

The nesting of a *least pre-fixpoint operator* (μ) inside a *greatest post-fixpoint operator* (ν) is what makes the completeness proof of CTL^f more than just an update from that of LTL. In the latter, the only 'eventuality' operator is **F**, which we could have defined as

$$\forall a, \mathbf{F}a = \mu x.(a \vee \mathbf{XF}x)$$

Looking back at Reynolds' algorithm, testing satisfiability of $\mathbf{F}\varphi$ amounts to checking if φ can be satisfied, and if not, test $\mathbf{F}\varphi$ in the next state, concluding negatively if the branch just loops without ever satisfying φ. This is because the fixpoint **F** is *constructive.*

Constructive operators in fair CTL

We define, given a transition system (S, R), a CTL^f-algebra which we consider canonic for the system, i.e. which transcribes in algebraic terms the definitions we gave in section 1 for semantics for CTL^f [2, Section 2].

Definition 12. Given a transition system (S, R), its *complex algebra* is the tuple

$$\mathbb{P}_{(S,R)} = (\mathcal{P}(S), \emptyset, S\backslash(-), \cup, \Diamond_R, EU_R, EG_R)$$

which is the classical Boolean algebra over the powerset of S, along with operators

$$\Diamond_R(a) = R^{-1}[a] = \{s \in S \mid \exists t \in a, sRt\}$$

and EU_R and EG_R defined as fixpoints (see above) so that we finally have a CTL^f-algebra.

Now, completeness becomes equivalent to proving that for every CTL^f-formula φ with variables in $\bar{p}$, if there exists a satisfying interpretation of φ in $\mathbb{P}_{(S,R)}$, then there exists one in every CTL^f-algebra A.

Indeed, $\mathbb{P}_{(S,R)}$ is designed so that a CTL^f-formula φ is semantically true if and only if there exists an interpretation of φ in $\mathbb{P}_{(S,R)}$ which satisfies φ. This is almost obvious from definitions for propositional cases and $\Diamond$; for EU and EG, we use the following lemmas:

Lemma 13. For every a, b in $\mathbb{P}_{(S,R)}$ it holds that

$$EU_R(a, b) = \bigcup_{i=0}^{\infty} D_n(a, b)$$

where D_n is defined inductively as

$$\begin{cases} D_0(a,b) = \emptyset \\ D_{n+1}(a,b) = a \cup (b \cap \Diamond_R D_n(a,b)) \end{cases}$$

Proof. (sketch)

- $\supseteq$ By induction on n we show $D_n(a,b) \subseteq EU_R(a,b)$.
- $\subseteq$ We show that $D(a,b) := \bigcup_{i=1}^{\infty} D_n(a,b)$ is a pre-fixpoint of $x \mapsto a \cup (b \cap \Diamond x)$. From the definition of EU the inclusion directly follows.

□

We can now state the corresponding lemma for EG:

Lemma 14. For every a, b in $\mathbb{P}_{(S,R)}$ it holds that $EG_R(a,b)$ is the set:

$$\{s \in S \mid \text{there exists an infinite path } s = s_0 R s_1 R \ldots \\ \text{such that } \forall k \in \mathbb{N}\ s_k \in a, \text{ and } s_j \in b \text{ for infinitely many } j \in \mathbb{N}\}$$

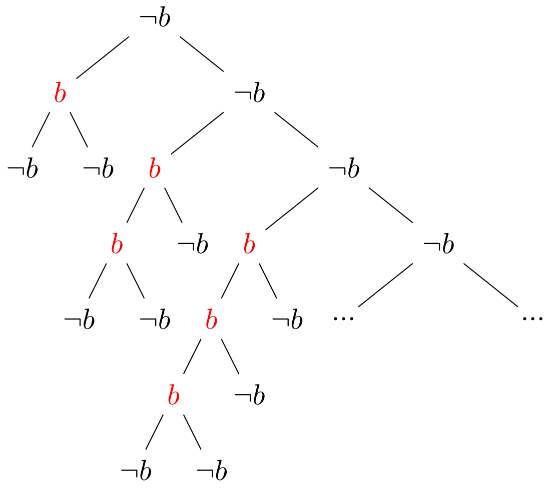
The operator EU is *constructive* (see [7, Intro]), because it can be described using pre-existing operators of the syntax; whereas the concept of 'infinite path' introduced by the operator EG cannot be reduced to anything else, so EG is not constructive. In particular, we do *not* have the equality:

$$EG_R(a,b) = \bigcap_{i=1}^{\infty} D'_n(a,b)$$

where

$$\begin{cases} D'_0(a,b) = \mathcal{P}(S) \\ D'_{n+1}(a,b) = a \cap \Diamond_R EU_R(b \cap D'_n(a,b), a) \end{cases}$$

Although the analogy is pretty clear (the sets are defined as an sequence induced by the 'fixpoint function' of the operator), it doesn't work because it fails to grasp infinite paths. For example, we can construct a tree belonging to every $D'_n(a,b)$ (for every n there is a finite path where every node is in a, and b contains at least n nodes), but such that no infinite path has the wanted property (of belonging to the set described in the last lemma). Informally, the following infinite binary tree is in every $D'_n(\mathcal{P}(S), b)$ but not in $EG_R(\mathcal{P}(S), b)$:



Ultrafilter computation

The proof of completeness for CTL^f in [2, Section 3] uses tableaux, but does not provide a way to compute a model for satisfiable formulas. The idea the proof starts with is fixing a general formula φ_0 which is *consistent* in a CTL^f-algebra $\mathbb{A}$ (not equal to $\bot$). The goal will be to prove that there exists a transition system and a valuation satisfying φ_0, therefore proving completeness. But the model is never *actually* computed. Because φ_0 is not equal to $\bot$, there exists an *ultrafilter* x_0 in $\mathbb{A}$ such that $\varphi_0 \in x_0$ [1, Section 3.3]. From there on, the specific tableaux method is unraveled, using the *ultrafilter frame* of $\mathbb{A}$; but we never really know what x_0 consists of, and what formulas it contains. This is the object of the research we present below.

Atomic lower bounds

In this subsection, what we try to do is characterize ultrafilters by atoms, as we did for unary LTL in 2.1. Not every infinite algebra is *atomic*, which means that maybe not every ultrafilter in $\mathbb{A}$ has a minimum. What is sure is that if there is a formula $\varphi_{0,m}$ such that $x_0 = \uparrow \varphi_{0,m}$, it must be absolutely restrictive on which trees can be accepted as models. Let us make precise what this means.

Definition 15. Let a be a consistent CTL^f-formula. We say that a formula a_m is an *atomic lower bound for a* if

1. $\bot < a_m \leq a$
2. for every CTL^f-formula ψ we have either $\psi \wedge a_m = a_m$, or $\psi \wedge a_m = \bot$.

Equivalently, a_m is an atomic formula under a.

For every p in $\bar{p}$, for every $n \geq 0$, we should have either $\Diamond^n p \wedge \varphi_{0,m} = \varphi_{0,m}$, or $\Diamond^n p \wedge \varphi_{0,m} = \bot$. In informal terms, either all models of $\varphi_{0,m}$ contain a n-th successor for the root whose coloring contains p, or none of them do. The same reasoning with $\Diamond^n \neg p$ leads us to say that for every n, for every p in $\bar{p}$,

$$\varphi_{0,m} \vdash \Box^n p \text{ or } \varphi_{0,m} \vdash \Box^n \neg p.$$

One way to restate this is by saying that colorings of models are determined by formula $\varphi_{0,m}$. Thus, we would like to construct this formula from φ_0 by fixing colorings which are not determined by it. An immediate problem is that formulas in the logic are finite, while trees are infinite. Even with the simplest of consistent formulas, $\varphi_0 = \top$, building $\varphi_{0,m}$ only with boxes is impossible, as there are infinitely many formulas $\bigwedge_{p \in \bar{p}} \bigwedge_{k=0}^{n} \Box^k p$ which are below $\top$. We need a formula which is below $\bigwedge_{p \in \bar{p}} \bigwedge_{k=0}^{n} \Box^k p$ for every n, and a fitting candidate is $\bigwedge_{p \in \bar{p}} AR(p, \bot)$, as

$$AR(p, \bot) = p \wedge \Box AR(p, \bot) = p \wedge \Box p \wedge \Box AR(p, \bot) = ...$$

Actually, we can define $\top_m = \bigwedge_{p \in \bar{p}} AR(p, \bot)$.

Proposition 16. For every CTL^f formula ψ, we have either $\top_m \wedge \psi = \bot$, or $\top_m \wedge \psi = \top_m$.

Proof. We proceed by induction over the structure of ψ. Note that due to the structure of $\top_m$, it is sufficient to show the property that for every $p \in \bar{p}$, either $AR(p, \bot) \wedge \psi = \bot$ or $AR(p, \bot) \wedge \psi = AR(p, \bot)$. For this report, we will focus on only one important point.

- let $\phi = EU(\psi, \chi)$. Suppose that the property holds for ψ and χ, by induction. Let p in $\bar{p}$. We first show that

$$AR(p, \bot) \wedge EU(\psi, \chi) = EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))$$

First,

$$\begin{aligned}
&EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot)) \\
&= [\psi \wedge AR(p, \bot)] \vee [\chi \wedge AR(p, \bot) \wedge \Diamond EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))] \\
&= AR(p, \bot) \wedge (\psi \vee [\chi \wedge \Diamond EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))]) \\
&\leq AR(p, \bot) \wedge (\psi \vee [\chi \wedge \Diamond EU(\psi, \chi)]) \text{ by monotonicity} \\
&= AR(p, \bot) \wedge EU(\psi, \chi).
\end{aligned}$$

On the other hand, we will show that

$$EU(\psi, \chi) \leq EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot)) \vee \neg AR(p, \bot).$$

By a least fixpoint argument using definition of $EU(\psi, \chi)$, a sufficient condition is

$$\begin{aligned}
&\psi \vee (\chi \wedge \Diamond[EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot)) \vee \neg AR(p, \bot)]) \\
&\qquad\qquad \leq EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot)) \vee \neg AR(p, \bot).
\end{aligned}$$

Equivalently, by switching back $AR(p, \bot)$ to the left side and by developing on the left, we only need to prove

$$\begin{aligned}
&[AR(p, \bot) \wedge (\psi \vee [\chi \wedge \Diamond EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))])] \\
&\qquad \vee [AR(p, \bot) \wedge \psi] \vee [\chi \wedge AR(p, \bot) \wedge \Diamond \neg AR(p, \bot)] \\
&\qquad\qquad \leq EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))
\end{aligned}$$

As the left member simplifies, this is still equivalent to

$$\begin{aligned}
&[AR(p, \bot) \wedge \psi] \vee [AR(p, \bot) \wedge \chi \wedge \Diamond EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))] \\
&\qquad\qquad \leq EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))
\end{aligned}$$

which is true by definition of EU. Therefore, the inequality is proven in both ways, which states our first equality. Now, by induction hypothesis, there are four cases, depending on the values of $AR(p, \bot) \wedge \psi$ and $AR(p, \bot) \wedge \chi$. In each case, it is straightforward to check that $EU(\psi \wedge AR(p, \bot), \chi \wedge AR(p, \bot))$ is either equal to $\bot$, or $AR(p, \bot)$, which concludes the proof for this point.

□

This definition of $\top_m$ is satisfactory, and we would like to extend this construction to all literals by defining

$$p_m := \top_m$$

$$(\neg p)_m := AR(\neg p, \bot) \wedge \bigwedge_{q \in \bar{p}, q \neq p} AR(q, \bot).$$

But here is the problem: while this definition for literals is the most natural from what we could guess, it does not fare well by induction. We cannot define $(\varphi_0 \wedge \varphi'_0)_m = \varphi_{0,m} \wedge \varphi'_{0,m}$.

Take for instance $\bar{p} = \{p, q\}$, $\varphi_0 = p$ and $\varphi'_0 = \neg q$: following the previous construction we have

$$\varphi_{0,m} \wedge \varphi'_{0,m} = AR(p, \bot) \wedge AR(q, \bot) \wedge AR(\neg q, \bot) = \bot.$$

This is why we stopped here in our search for atomic lower bounds of CTL^f formulas. However, using automata to compute the set of *all* atoms beneath φ_0 might be the next step towards actually computing ultrafilters, or finding another constructive solution to the SAT problem for CTL^f. This is left to future work.

Acknowledgments

I would like to thank Adrien Guatto of IRIF for his advice on the implementation in OCaml, and the FSMP for financing this internship. Most of all, I am grateful for my supervisor's availability, insight and support during these five months.

References

[1] M. Gehrke and S. van Gool, *Topological duality for distributive lattices, and applications.* arXiv, 2022. [Online]. Available: https://arxiv.org/abs/2203.03286v1

[2] S. Ghilardi and S. van Gool, "Monadic second order logic as the model companion of temporal logic," in *Proceedings of the Thirty first Annual IEEE Symposium on Logic in Computer Science (LICS 2016)*, 2016.

[3] ——, "A model-theoretic characterization of monadic second order logic on infinite words," *The Journal of Symbolic Logic*, vol. 82, no. 1, 2017.

[4] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977.

[5] B. Subrata, "Propositional logic," 09 2017. [Online]. Available: https://www.researchgate.net/publication/319702897_Propositional_Logic

[6] M. Reynolds, "A traditional tree-style tableau for LTL," 2016. [Online]. Available: https://arxiv.org/abs/1604.03962

[7] L. Santocanale, "Completions of mu-algebras," 2005. [Online]. Available: https://arxiv.org/abs/math/0508412

[8] M. Bertello, N. Gigante, A. Montanari, and M. Reynolds, "Leviathan: A new LTL satisfiability checking tool based on a one-pass tree-shaped tableau," in *IJCAI*, 2016.

[9] L. Geatti, N. Gigante, and A. Montanari, "Black: A fast, flexible and reliable LTL satisfiability checker," ser. CEUR Workshop Proceedings, vol. 2987. CEUR-WS, 2021, p. 6.

[10] F. Laroussinie, "Intro to the mpri 2.82 course 'modelisation and verification of timed systems'," https://www.irif.fr/ francoisl/DIVERS/mpri2122intro.pdf, 2021-2022.

[11] J. Cristau, "Automata and temporal logic over arbitrary linear time," *CoRR*, vol. abs/1101.1731, 2011. [Online]. Available: http://arxiv.org/abs/1101.1731

[12] S. Demri and P. Gastin, "Specification and Verification using Temporal Logics," in *Modern applications of automata theory*, D'Souza, Deepak, Shankar, and Priti, Eds. World Scientific, 2012, vol. 2, pp. 457–494. [Online]. Available: https://hal.inria.fr/hal-00776601

[13] P. Gastin and D. Oddoux, "Fast ltl to büchi automata translation," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[14] P. Wolper, "The tableau method for temporal logic: An overview," *Logique et Analyse*, vol. 28, no. 110/111, pp. 119–136, 1985. [Online]. Available: http://www.jstor.org/stable/44084125

[15] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis," *Electronic Notes in Theoretical Computer Science*, vol. 262, pp. 113–125, 2010, proceedings of the 6th

Workshop on Methods for Modalities (M4M-6 2009). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066110000319