Bounded model checking and passive learning of LTL formulas

THOMAS XU
(supervisor: ADRIEN POMMELLET)

Technical Report n^ooutput, January 2025
revision

Linear Temporal Logic (LTL) is a formalism widely used for specifying and verifying properties of reactive systems over time. Unlike CTL (Computation Tree Logic), which allows branching structures, LTL follows a linear-time perspective, making it suitable for reasoning about sequences of system states. A Kripke structure serves as a foundational model for both CTL and LTL, representing systems as states and transitions labeled with atomic propositions. Bounded Model Checking (BMC) has emerged as an efficient technique to verify LTL properties by searching for a counterexample with bounded length, while passive learning techniques aim to infer LTL formulas from observed system behaviors. In practical, we will have two samples of Kripke structures P and N, such that we want to find an LTL formula f that it is verified by elements of P and contradicted by elements of N. Once f is found, BMC will be used to verify the correctmess of f. Our objective is to extend an existing tool—currently supporting CTL for LTL verification and compare the performance of finding a formula using both logics. This comparison will help determine whether LTL or CTL is faster and more effective in specific verification tasks, providing insights into their practical efficiency.

La logique temporelle linéaire (LTL) est un formalisme répandu pour la spécification et la vérification de propriétés des systèmes réactifs dans le temps. Contrairement à CTL (Computation Tree Logic) qui autorise les structures branchantes, LTL suit une perspective temporelle linéaire ce qui la rend adaptée au raisonnement sur les séquences d'états d'un système. Une structure de Kripke sert de modèle fondamental pour la CTL et la LTL, représentant les systèmes sous forme d'états et de transitions étiquetés avec des propositions atomiques. Le Model Checking Borné (BMC) a émergé comme une technique efficace pour vérifier les propriétés LTL en recherchant un contre-exemple de longueur bornée, tandis que les techniques d'apprentissage passif visent à inférer des formules LTL à partir des comportements observés du système. En pratique, on va avoir deux échantillons de structure de Kripke P et N, le but étant de trouver une formule LTL f telle que, f est vérifiée par chaque élément de P et rejetée par chaque structure de N. Une fois f trouvée, le BMC va nous permettre de vérifier que f est une formule valide pour nos systèmes. Notre objectif est d'étendre un outil existant – qui prend actuellement en charge la CTL – pour y inclure la vérification LTL, et de comparer les performances de recherche d'une formule en utilisant les deux logiques. Cette comparaison permettra de déterminer si la LTL ou la CTL est plus rapide et efficace dans des tâches de vérification spécifiques, offrant ainsi des perspectives sur leur efficacité pratique.

Keywords
Automata, Kripke structure, LTL, Bounded model checking,

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
thomas.xu@lre.epita.fr – http://www.lre.epita.fr/

Chapter 1

Introduction

Many real-world systems we use every day can be seen as reactive systems that interact with their environment. For reactive systems, the correctness depends on the executions of the system across time not only the inputs and outputs. Temporal logic provides mathematically precise notations to express those properties while being intuitive because of its proximity with natural language. *Linear Temporal Logic (LTL)* is a very good formalism for reasoning about the temporal behavior of systems. It provides a robust framework for specifying properties such as safety, liveness, and fairness, making it a cornerstone of formal verification in fields ranging from software engineering to hardware design. LTL's expressiveness and intuitive syntax enable practitioners to articulate temporal properties that capture the essence of desired system behaviors, such as "a request is eventually followed by a response."

In many real-world scenarios, the properties that describe system behavior are not readily available. Instead, one may only have access to execution traces examples of system runs that demonstrate desired or undesired behavior. *Passive learning* is a set of methods that computes a theorical model of a system from a given set of data, without any furthur queries. By analyzing these examples, passive learning seeks an LTL formula that captures the essence of the observed behavior while distinguishing between acceptable and unacceptable executions. This approach provides critical insights into system functionality and facilitates understanding without requiring an explicit specification.

Once such a formula is computed, we have to check whether it is correct in our context, and that can be done with model checking. Standardly in model checking, the studied system will be modeled as a finite state machine and the states will be traversed to check the properties. If the system does not verify the property, then a counterexample will be given to us. Its biggest issue is the fact that it does not scale well with the size of a system since it needs a lot of memory. Here is where *Bounded model checking (BMC)*[5] comes handy. BMC explores the state space of a system within a specified bound k and while it does not completely solve the memory issue of the standard model checking, it does solve some cases that cannot be solved with standart model checking. We will see later how to can translate a BMC problem to a propositionnal satisfiability problem that can leverage modern SAT solvers that are able to handle a lot of variables without suffuring from the space explosion problem.

Ultimately, our goal is to check whether it is faster to learn a LTL or a CTL formula in a passive scenario knowing that both are NP-hard[4][2]. We have a C++ library that does it for CTL, the original objective of this work was to implement the passive learning of LTL formulas in this library and the bounded model checking to ensure that the computed formula is verified by the systems that had to verify it and violated by the others.

Chapter 2

Preliminary Definitions

2.1 Kripke structures[1]

2.1.1 Definition

A **Kripke structure** is a mathematical model commonly used to represent systems in formal verification, particularly in model checking. It provides a framework to describe the behavior of a system in terms of states and transitions. Formally, a Kripke structure is defined as:

A **Kripke structure K** is a tuple $M = \langle Q, I, AP, T, L \rangle$, where:

- Q: A finite set of states.
- $I \subseteq Q$: A set of initial states.
- AP: A finite set of atomic propositions.
- $T \subseteq Q \times Q$: A transition relation between states, representing the system's behavior.
- $L : Q \to 2^{AP}$: A labelling function.

Kripke structures are the backbone of many formal verification techniques, as they allow systematic exploration of all possible system behaviors. To define the sequencial behavior of a Kripke structure M, we use paths. A path π in M is a sequence $\pi = (s_1, ..., s_n)$ of states, given an order that respects the transition relation T of M. That is, $T(s_i, s_{i+1})$ for all $0 \leq i < |\pi| - 1$. If s_0 is an initial state (written $I(s_0)$) then we say that the path is initialized.

2.1.2 Example: Traffic Light Model

Consider the modeling of a traffic light system as a Kripke structure. The traffic light can be in one of three states: green, yellow, or red. We can say that G = 100, Y = 010 and R = 001, where each bit would correspond to a certain light color to simplify notations. Each state transitions to the next in a cyclic order, as shown below.

The Kripke structure representing the traffic light system is defined as:

- $Q = \{G, Y, R\}$: The states, corresponding to green, yellow, and red lights.
- $I = \{G\}$: The initial state, where the light starts as green.
- $AP = \{$"green", "yellow", "red"$\}$: The atomic propositions indicating the light's color.

- $T = \{(G, Y), (Y, R), (R, G)\}$: The transition relation between states.

The state diagram for this Kripke structure is illustrated below:

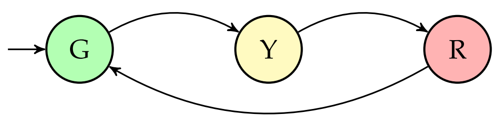


Figure 2.1: Kripke structure modelization of traffic lights.

2.1.3 Kripke Structures in Model Checking

In model checking, systems are modeled as Kripke structures to allow formal reasoning about their behaviors. Properties of interest, such as safety or liveness, are specified using temporal logics like Linear Temporal Logic (LTL). The model checker systematically explores the states and transitions in the Kripke structure to verify whether the specified properties hold.

2.2 LTL[1]

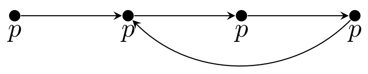
For reactive systems, the correctness depends on the executions of the system across time not only the inputs and outputs. To capture those properties we will use temporal logic that provides mathematically precise notations to express those properties while being intuitive because of its proximity with natural language. The underlying nature of time in temporal logics can be either linear or branching. The term "temporal" refers to the order of events (i.e "Receive a response after sending a request") and not the precise time of events (i.e "Receive a response 5 seconds after sending a request"). In terms of transition systems, neither the duration of taking a transition nor state residence times can be specified using the elementary modalities of temporal logics. Instead, these modalities do allow for specifying the order in which state labels occur during an execution, or to assess that certain state labels occur infinitely often in a (or all) system execution. One might thus say that the modalities in temporal logic are time-abstract.

In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. In our case, we will consider LTL (Linear Temporal Logic), a temporal logic that is based on a linear-time perspective.

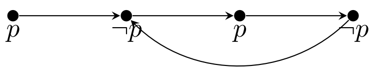
2.2.1 Definition

LTL is an extension of propositional logic, so it inherits boolean variables, operators ($\wedge \vee \neg$) to which we add some temporal operators (**X,F,G,U**). If all initialized paths of a Kripke structure satisfy a property, we say that the property holds for the Kripke structure.
X is the next operator, the formula **Xp** specifies that the property holds at the next step.
For example, **Xp** holds on figure a[2.2a] but not on b[2.2b].

The next class of temporal operators that we discuss, allows specifying repeated unbounded behavior along an infinite path.



(a) Figure a



(b) Figure b

The Globally operator **G** is generally used for safety properties (verify that something is always true or false). A formula **Gp** holds along a path if p holds in all states of the path.
For example the formula **Gp** is hold for the system a[2.2a] while it is violated by the system b[2.2b] at its second state.

The Finally operator **F** is generally used for liveness properties (verify that something happens at a point of time). The formula **Fp** holds along a path if p holds somewhere on the path. Equivalently, it fails to hold if p stays unsatisfied along the whole path.
For instance **F**$\neg p$ is hold on figure b[2.2b] but not figure a[2.2a] since it does not contain a state that holds $\neg p$.

The Until operator requires two LTL formulas p and q, **pUq** holds on a path if there is a point in the future where q holds and p holds at all moments until that point.
For instance **pU**$\neg p$ holds for b and not for a.

We can also combine them to obtain new temporal modalities. For instance **FGp** ("eventually forever") describes the property that from a moment i on, p will hold for every successor states, **GFp** ("infinitely often") means that at each moment i, there exist j such that $i < j$ at which p is hold. For instance, in the case of traffic lights[2.1], some properties that it must verify are:
-**GF 100**: At all point, there will be a future moment where the light turns green.
-$\neg FG$ **100**: There is no point during the execution from when, the light will remain green infinitely.
From there, we can construct formulas using those operators as well as boolean connectors to design more complex specifications for our systems.

2.2.2 Semantics

The formal semantics of temporal formulas is defined with respect to paths of a Kripke structure. Let π be an infinite path of a Kripke structure M and let f be a temporal formula. We define recursively when f holds on π, written $\pi \models f$, as follows:

$$\begin{aligned}
\pi \models p &\iff p \in L(\pi(0)) \\
\pi \models \neg f &\iff \pi \not\models f \\
\pi \models f \wedge g &\iff \pi \models f \text{ and } \pi \models g \\
\pi \models Xf &\iff \pi_1 \models f \\
\pi \models Gf &\iff \pi_i \models f \text{ for all } i \geq 0 \\
\pi \models Ff &\iff \pi_i \models f \text{ for some } i \geq 0 \\
\pi \models f\,U\,g &\iff \pi_i \models g \text{ for some } i \geq 0 \text{ and } \pi_j \models f \text{ for all } 0 \leq j < i \\
\pi \models f\,R\,g &\iff \pi_i \models g \text{ if for all } j < i, \pi_j \not\models f
\end{aligned}$$

The semantics of the other Boolean operators such as disjunction and implication can be inferred from the above definitions. As mentioned above, we say that a temporal formula f holds for a Kripke structure M, written $M \models f$, if and only if $\pi \models f$ for all initialized paths π of M.

Finally, we say that two temporal formulas f and g are equivalent, written $f \equiv g$, if and only if $M \models f \leftrightarrow M \models g$ for all Kripke structures M. With this notion, the semantics imply that:

$$\neg F \neg p \equiv Gp.$$

Thus, F and G are dual operators.

Chapter 3

Model checking

During the design process of complex software and hardware systems, verification often consumes more resources than construction. Efforts are continuously being made to reduce the burden of verification while improving its effectiveness and coverage. Formal methods have emerged as a powerful solution, enabling early integration of verification into the design process, enhancing verification techniques, and significantly reducing the time required. Simply put, formal methods are "the applied mathematics of modeling and analyzing ICT systems." Their goal is to ensure system correctness through mathematical precision. Over the past two decades, advances in formal methods have led to the creation of effective verification techniques that enable the early detection of defects. These techniques are supported by sophisticated software tools that automate many verification steps, making the process more efficient and reliable. Model-based verification is a part of formal methods, it creates accurate models that mathematically describe all possible behaviors of a system. Interestingly, the process of building these models often uncovers flaws such as missing details, ambiguities, and inconsistencies in informal specifications. The verification technique we will study is the model checking. Model checking is a verification technique that explores all possible system states in a brute-force manner so that it can show if a property if verified by a certain system. It is usually used to verify two types of properties, safety property declares what should happen (or what should never happen) and liveness property specifies what should eventually happen. A counterexample to a safety property would be a trace of states where the last state violates the property and for the liveness property, it would be a path to a loop that does not contain the desired state.

3.1 Standard model checking

3.1.1 Model checking process

To apply model checking to a design, there are three different phases:

- Modeling phase: models the system under consideration using a Kripke structure and formalize the property to be checked using LTL. The model should describe the desired behavior in an accurate and unambiguous way, states containing current values of variables and transitions describe how the system evolves from a state to another.
- Running phase: runs the model checker to assess the validity of the property for the system.

- Analysis phase: If the property is violated then generate a counterexample by simulation, then refine the model and repeat the process. If the model is too big to handle then multiple methods are available that try to abstract the system while preserving the validity of the property. Else the models verifies the property and it ends here.

3.1.2 Example

Let us consider the traffic light to which we will add a new state O (with valuation 0001) such that the models goes there if it is out of order.

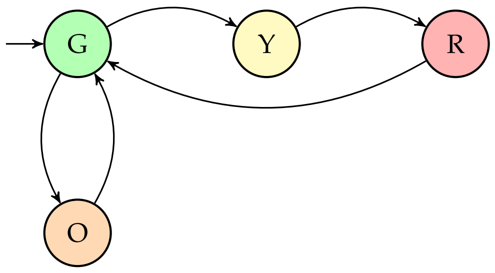


Figure 3.1: New traffic light.

Let M be the Kripke structure modeling the behavior described above. The following properties are properties we want it to satisfy and we can compute the result using model checking. The standard technique for model checking LTL is to compute the product of the Kripke structure with an automaton that represents the negation of the property (this automaton captures exactly the execution sequences that violate the LTL formula). Emptiness of the product automaton is an evidence of the correctness of the property.
Liveness properties:

- $M \models GFgreen..$
- $M \models GFyellow.$
- $M \models GFred.$

This shows that our system is indeed capable at each point of time, to display any colors of a traffic light.
Safety properties:

- $M \models \neg Ggreen.$
- $M \models \neg Gyellow.$
- $M \models \neg Gred.$
- $M \not\models G\neg orange.$

Our system is not stuck infinitely on any colors but it can go to the undesired state O violating $G\neg orange.$

3.1.3 Pros and cons

Strengths of Model Checking

- **Versatile application**: Model checking is applicable to various domains, including embedded systems, software engineering, and hardware design.
- **Partial verification**: Properties can be checked individually, allowing developers to focus on critical requirements without needing a complete specification.
- **Error robustness**: Unlike testing and simulation, model checking does not rely on the likelihood of error exposure and systematically analyzes the system.
- **Debugging aid**: Diagnostic feedback is provided when a property is invalidated, making it valuable for debugging purposes.
- **Ease of use**: As a potential "push-button" technology, model checking requires minimal user interaction and expertise, making it accessible to a broader audience.
- **Industrial adoption**: Increasing interest from industries has led to in-house verification labs, growing job opportunities, and the availability of commercial tools.
- **Seamless integration**: It can be incorporated into existing development cycles, with a manageable learning curve and potential for shorter development times.
- **Mathematical foundation**: The method is grounded in robust theoretical principles, including graph algorithms, data structures, and logic.

Weaknesses of Model Checking

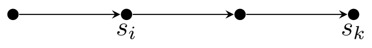
- **Limited to control-intensive applications**: Model checking is less suited for data-intensive applications, where data often spans infinite domains.
- **Decidability constraints**: For infinite-state systems or abstract data types requiring undecidable or semi-decidable logics, model checking may not be feasible.
- **Model dependency**: Verification applies to the system model, not the actual product or prototype. Complementary techniques, like testing, are needed to address fabrication faults or coding errors.
- **Incomplete property coverage**: Only explicitly stated requirements are verified, leaving unchecked properties unvalidated.
- **State-space explosion**: Accurately modeling large systems can result in state spaces exceeding available memory, even with advanced reduction techniques.
- **Abstraction expertise required**: Developing smaller, manageable models and formulating properties in logical formalism often demand significant expertise.
- **Potential tool errors**: Like any software, model checkers can contain bugs, though advanced procedures have been formally verified using theorem provers.

3.2 Bounded model checking[5]

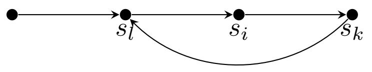
As we mentioned before, this method does not solve the explosion of the state space but still manages to solve cases that standard model checking cannot. The original motivation behind this method was to leverage the success of SAT in solving boolean formulas to model checking. For the past decades, there has been a tremendous increase of SAT solvers' power making them able to work with hundreds of thousands of variables and millions of clauses.
That being said, the general idea is to find a counterexample of size bounded by k, do to so, we will consider finite prefixes of paths that may be a witness to an existential model checking problem.
And since LTL formulas are defined over all paths, finding a counterexample is equivalent to finding a trace that contradicts them. Such trace is called a witness for the property. For example, let M be our traffic light with the orange state, finding a counterexample to **G**$\neg$*Orange* is equivalent to answering whether there is a trace that is a witness for **F***Orange*.
In the original paper[5], the authors use path quantifiers **E** and **A** to denote whether the formula is expected to be correct on all paths or only some paths. For a kripke structure M and a LTL formula f, $M \models$ **E**f means that there exists an initialized path that verifies the property and $M \models Af$ means that M verifies f for each initialized path. In our case, since the objective is to find a counterexample, we will only use **E**. We will assume that the formula is given in negation normal form (NNF), in which negations are only allowed just before atomic propositions and every LTL formula can be put in this form using the duality of LTL operators and De-Morgans's laws.

Paths

As we are operating on prefixes of paths, it is crucial to observe that even though the prefix is finite, it can represents an infinite path if there is a *back loop* from the a state to the prefix of any previous states as shown in 3.2b. If we can't find such back loop 3.2a, then the prefix does not tell us anything about the infinite behavior of the path after the state s_k.
For example, if a prefix does not contain a back loop then it cannot be a witness for **G**p, even if p holds along all states from s_0 to s_k, p might not hold for state s_{k+1} so we cannot conclude that we have found a witness.



(a) No loop



(b) (k,l)-loop

Figure 3.2: Examples of bounded paths: no loop and (k,l) loop

Definition 1 ((k,l)-loop) *For $l \leq k$, we call a path π a (k,l)-loop if $T(\pi(k), \pi(l))$ and $\pi = u \cdot v^\omega$, where $u = (\pi(0), \ldots, \pi(l-1))$ and $v = (\pi(l), \ldots, \pi(k))$. We call π a k-loop if there exist $k \geq l \geq 0$ for which π is a (k,l)-loop.*

We introduce the concept of k-loops to define the bounded semantics of model checking, which refers to the semantics of model checking constrained to bounded traces. This bounded semantics serves as an approximation of the unbounded semantics, enabling us to define the bounded model checking problem. In the following section, we present a method to translate a bounded model checking problem into a satisfiability problem.

Bounded semantics

Under the bounded semantics, only a finite prefix of a path is considered. Specifically, the first $k+1$ states $(s_0, \ldots, s_k)$ of a path are used to evaluate the validity of a formula along that path. For paths that are k-loops, the original LTL semantics are preserved, as the finite prefix of length k contains all the necessary information about the infinite path.

Definition 2 (Bounded semantics for loops) *Let $k \geq 0$ and π a k-loop, a LTL formula f is valid along the path π with bound k (written $\pi \models_k f$) iff $\pi \models f$.*

Now, let us consider the case where π is not a k-loop. In the unbounded semantics, the formula f=**F**p is valid along π if there exists an index $i \geq 0$ such that p holds on the suffix π_i of π. However, in the bounded semantics, the $(k+1)$-th state $\pi(k)$ does not have a successor. As a result, unlike the unbounded case, the bounded semantics cannot be defined recursively over the suffixes π_i of π.

To address this, we introduce the notation $\pi \models^i_k f$, where i denotes the current position in the prefix of π. This notation means that the suffix π_i of π satisfies f, i.e., $\pi \models^i_k f$ implies $\pi_i \models f$.

Definition 3 (Bounded semantics for loops) *Let $k \geq 0$ and π is not a k-loop, a LTL formula f is valid along the path π with bound k (written $\pi \models_k f$) iff $\pi \models^0_k f$ where*

$$\begin{aligned}
\pi \models^i_k p &\iff p \in L(\pi(i)),\\
\pi \models^i_k \neg p &\iff p \notin L(\pi(i)),\\
\pi \models^i_k f \wedge g &\iff \pi \models^i_k f \textit{ and } \pi \models^i_k g,\\
\pi \models^i_k f \vee g &\iff \pi \models^i_k f \textit{ or } \pi \models^i_k g,\\
\pi \models^i_k Gf &\textit{ is always false}\\
\pi \models^i_k Ff &\iff \exists j,\ i \leq j \leq k \textit{ such that } \pi \models^j_k f,\\
\pi \models^i_k Xf &\iff i < k \textit{ and } \pi \models^{i+1}_k f,\\
\pi \models^i_k f\,U\,g &\iff \exists j,\ i \leq j \leq k \textit{ such that } \pi \models^j_k g \textit{ and } \forall n,\ i \leq n < j,\ \pi \models^n_k f,\\
\pi \models^i_k f\,R\,g &\iff \exists j,\ i \leq j \leq k \textit{ such that } \pi \models^j_k f \textit{ and } \forall n,\ i \leq n < j,\ \pi \models^n_k g.
\end{aligned}$$

Note that if π is not a k-loop, then Gf is not valid along π in the bounded semantics with bound k, since f might not hold along π_{k+1}. These constraints imply that, in the bounded semantics, the duality between G and F (i.e., $\neg Ff \equiv G\neg f$) no longer holds.

Next, we describe how the existential model checking problem ($M \models_k \mathbf{E}f$) can be reduced to a bounded existential model checking problem ($M \models_k \mathbf{E}f$). The foundation for this reduction is provided by the following two lemmas.

Lemma 1 *Let f be an LTL formula and π a path. Then:*

$$\pi \models_k f \implies \pi \models f.$$

Lemma 2 *Let f be an LTL formula and M a Kripke structure. If $M \models$ **E**f, then there exists $k \geq 0$ such that:*

$$M \models_k \boldsymbol{E}f$$

Based on Lemmas 1 and 2, we can now state the main theorem of this section. Informally, Theorem 1 establishes that if we select a sufficiently large bound, the bounded and unbounded semantics become equivalent.

Theorem 1 *Let f be an LTL formula and M a Kripke structure. Then:*

$$M \models \boldsymbol{E}f \iff \exists k \geq 0 \text{ such that } M \models_k \boldsymbol{E}f.$$

3.2.1 Reducing bounded model checking to SAT

After defining the semantics for bounded model checking, we will show how to reduce BMC to a problem of propositional satisfiability which allows us to use efficient SAT solvers to perform model checking.

Given a Kripke structure M, an LTL formula f, and a bound k, we construct a propositional formula $[\![M, f]\!]_k$. Let $s_0, s_1, \ldots, s_k$ be a finite sequence of states on a path π. Each s_i represents a state at time step i, consisting of an assignment of truth values to the set of state variables.

The formula $[\![M, f]\!]_k$ encodes constraints on $s_0, s_1, \ldots, s_k$ such that $[\![M, f]\!]_k$ is satisfiable iff π serves as a witness for f. It has three components:

- The propositional formula $[\![M]\!]_k$ that constrains $s_0, ..., s_k$ to be a valid initialized path.
- The loop condition which is a propositional formula that is evaluated to true if π contains a loop.
- The propositional formula that constrains π to satisfy f.

Definition 4 (Unfolding of the transition relation) *For a Kripke structure M and $k \geq 0$,*

$$[\![M]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}).$$

Let us consider the example in 3.1 where each state s of the system M is represented by four bits variables (s[0], s[1], s[2] and s[3]) where the first bit corresponds to traffic light displaying green and the other bits respectively yellow, red and orange. Let us consider the case where k = 2, these are the results of unrolling the transition relation:
The initial state is represented by:

$$I(s) := s[0] \wedge \neg s[1] \wedge \neg s[2] \wedge \neg s[3].$$We will write this s[1] to make it more readable.

The transition relation is represented by:

$$T(s, s') := (s[0] \wedge s'[1]) \vee (s[1] \wedge s'[2]) \vee (s[2] \wedge s'[0]) \vee (s[0] \wedge s'[3]) \vee (s[3] \wedge s'[0])$$

And ultimately:

$$[\![M]\!]_2 := I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2).$$

The shape of the path π will have an influence on the translation of a LTL formula, so we define a propositional formula ${}_lL_k$ to be true iff there is a transition from state s_k to s_l.

Definition 5 (Loop condition) *The loop condition ${}_lL_k$ is true iff there exists a back loop from the state s_k to a previous state or itself: $L_k := \bigvee_{l=0}^{k} {}_lL_k$.*

Depending on whether a path is a k-loop, we have two distinct translations of a temporal formula f. First, we consider the case where the path π is a k-loop. For such a path π, we provide a recursive translation of the LTL formula f. This translation processes the formula by recursively evaluating its subterms and states within π. The intermediate formula ${}_l[\![.]\!]_k^i$ depends on three parameters: l for the start position of the loop, k for the bound, and i for the current position in π.

Definition 6 (Successor in a Loop) *Let k,l,i such that $l, i \leq k$, we define the successor $succ(i)$ of i in a (k,l)-loop as $succ(i) := i + 1$ if $i < k$ and $succ(i) := l$ if $i = k$.*

Definition 7 (Translation of a LTL formula for a loop) *Let f, g be two LTL formulas, $k, l, i \geq 0$ with $i, l \leq k$.*

$$\begin{aligned}
{}_l[\![p]\!]_k^i &:= p(s_i)\\
{}_l[\![\neg p]\!]_k^i &:= \neg p(s_i)\\
{}_l[\![f \wedge g]\!]_k^i &:={}_l [\![f]\!]_k^i \wedge_l [\![g]\!]_k^i\\
{}_l[\![f \vee g]\!]_k^i &:={}_l [\![f]\!]_k^i \vee_l [\![g]\!]_k^i\\
{}_l[\![\boldsymbol{X}f]\!]_k^i &:={}_l [\![f]\!]_k^{succ(i)}\\
{}_l[\![\boldsymbol{G}f]\!]_k^i &:={}_l [\![f]\!]_k^i \wedge_l [\![\boldsymbol{G}f]\!]_k^{succ(i)}\\
{}_l[\![\boldsymbol{F}f]\!]_k^i &:={}_l [\![f]\!]_k^i \vee_l [\![\boldsymbol{F}f]\!]_k^{succ(i)}\\
{}_l[\![f\boldsymbol{U}g]\!]_k^i &:={}_l [\![g]\!]_k^i \vee ({}_l[\![f]\!]_k^i \wedge_l [\![f\boldsymbol{U}g]\!]_k^{succ(i)})
\end{aligned}$$

In this definition, a new propositional variable is introduced for each intermediate formula ${}_l[\![h]\!]_k^i$, where h is a subterm of the LTL formula f and i ranges from 0 to k. The number of variables is $O(|f| \times k)$ and the size of the propositional formula ${}_l[\![f]\!]_k^0$ is also $O(|f| \times k)$.

In cases where the path π is not a k-loop, the translation can be handled as a special case of the k-loop translation. For Kripke structures with total transition relations, every finite path π can be extended into an infinite path. Given that the properties beyond state s_k are unknown, a conservative approximation is made, assuming all properties after s_k are false.

Definition 8 (Translation of a LTL formula without a loop) *Inductive case: $\forall i \leq k$:*

$$\begin{aligned}
{}_l[\![p]\!]_k^i &:= p(s_i)\\
{}_l[\![\neg p]\!]_k^i &:= \neg p(s_i)\\
{}_l[\![f \wedge g]\!]_k^i &:={}_l [\![f]\!]_k^i \wedge_l [\![g]\!]_k^i\\
{}_l[\![f \vee g]\!]_k^i &:={}_l [\![f]\!]_k^i \vee_l [\![g]\!]_k^i\\
{}_l[\![\boldsymbol{X}f]\!]_k^i &:={}_l [\![f]\!]_k^{i+1}\\
{}_l[\![\boldsymbol{G}f]\!]_k^i &:={}_l [\![f]\!]_k^i \wedge_l [\![\boldsymbol{G}f]\!]_k^{i+1}\\
{}_l[\![\boldsymbol{F}f]\!]_k^i &:={}_l [\![f]\!]_k^i \vee_l [\![\boldsymbol{F}f]\!]_k^{i+1}\\
{}_l[\![f\boldsymbol{U}g]\!]_k^i &:={}_l [\![g]\!]_k^i \vee ({}_l[\![f]\!]_k^i \wedge_l [\![f\boldsymbol{U}g]\!]_k^{i+1})
\end{aligned}$$

Base case:

$$[\![f]\!]_k^{k+1} := false$$

We can combine all three components to encode the BMC problem in propositional logic as follows.

Definition 9 (General translation) *Let f be a LTL formula, M a Kripke structure and $k \geq 0$*

$$[\![M, f]\!]_k = [\![M]\!]_k \wedge ((\neg L_k \wedge [\![f]\!]_k^l) \vee \bigvee_{l=0}^{k} ({}_lL_k \wedge_l [\![f]\!]_k^0))$$

In the disjunction, the left side is the case where there is no loop back and uses the translation without loop. The right side represents all starting points l of a loop and the translation for a (k,l)-loop is conjoined with the associated ${}_lL_k$ loop condition. The size of this formula is $O(|f| \times k \times |M|)$ where $|M|$ is the size of the syntactic description of the inital state i and the transition relation T.

Theorem 2 $[\![M, f]\!]_k$ *is satisfiable iff* $M \models_k f$.

Example 1 *Let us consider the example in 3.1, we do not want it to reach the state O which can be represented by the formula* **G**$\neg p$ *where p = s[3]. Using BMC, we attempt to find a counterexample for this property which is equivalent to search for a witness of* **F**p*. If such witness exists, then the traffic lights can reach this state so the property is violated by our system. However, if we cannot find a suitable witness then the property holds up for the given bound k.*

Let us consider the case where k = 2, we already computed the unrolling of the transition relation3.2.1:

$$[\![M]\!]_2 := I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2).$$

The loop condition is:

$$L_2 := \bigvee_{i=0}^{2} T(s_2, s_l)$$

The translation for paths without loops is:

$$\begin{aligned}
[\![Fp]\!]_2^0 &:= p(s_0) \vee [\![Fp]\!]_2^1 \\
[\![Fp]\!]_2^1 &:= p(s_1) \vee [\![Fp]\!]_2^2 \\
[\![Fp]\!]_2^2 &:= p(s_2) \vee [\![Fp]\!]_2^3 \\
[\![Fp]\!]_2^3 &:= 0 \\
[\![Fp]\!]_2^0 &:= p(s_0) \vee p(s_1) \vee p(s_2)
\end{aligned}$$

The translation with loops can be done the same way. We can now put it together to get the propositional formula

$$[\![M, Fp]\!]_2 = [\![M]\!]_2 \wedge \left((\neg L_k \wedge [\![Fp]\!]_2^l) \vee \bigvee_{l=0}^{2} ({}_lL_2 \wedge_l [\![Fp]\!]_2^0)\right)$$

Since a single trace to a bad state is enough for falsifying a safety property, we can omit the loop condition in the formula above which gives us:

$$[\![M, Fp]\!]_2 = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge p(s_0) \wedge p(s_1) \wedge p(s_2)$$

$[\![M, Fp]\!]_2$ is satisfied by the path 1000, 0001, 1000 which is then a path that violates the safety property.

Chapter 4

Passive learning

In many real-world scenarios, the properties that describe system behavior are not readily available. Instead, one may only have access to execution traces examples of system runs that demonstrate desired or undesired behavior. *Passive learning* is a set of methods that computes a theorical model of a system from a given set of data, without any furthur queries. By analyzing these examples, passive learning seeks an LTL formula that captures the essence of the observed behavior while distinguishing between acceptable and unacceptable executions. This approach provides critical insights into system functionality and facilitates understanding without requiring an explicit specification. In practice, we have 2 samples of Kripke structures P and N and we seek a LTL formula f such that every element of P verify f and every element of N violate f. The paper [6] introduces two methods to do that. One of those is based on SAT and tries to reduce the problem of learning a LTL formula to a satisfiability problem to benefit from efficient SAT solvers. It has 3 essential features, first, it learns LTL formulas of minimal size which is easier to read and understand. Second, once we have learned a formula, we can query for more distinct formulas that are consistent with the sample, which can give explanations on what is happening in reality. Lastly, the algorithm does not rely on an a priori given set of templates, so it is not restricted to a fixed class of templates.

Implementing this was the main purpose using the SAT based learning algorithm ([6] for more details) and then verify that the learned formula was correct using BMC but since we are falling behind schedule by a lot, we will skip that part here.

Chapter 5

Implementation

5.1 Work done

We implemented everything mentioned earlier in LearnCTL using C++20, a library of Adrien Pommelet, a researcher at LRDE. This library implements the passive learning of a Computation Tree Logic (CTL) formula that separates a set of positive and negative Kripke structures, i.e. is verified by the former but not the latter. As we have mentionned it, since we fell behind the schedule, we decided to focus on implementing bounded model checking in priority. For now, we will try to implement a simple version where we only consider no-looping semantic and will use for bound k, the recurrence diameter for reachability which is the longest loop-free path in M starting from an initial state (more optimizations can be found in [6]).

5.2 Result

We don't have any result to show so far.

Chapter 6

Conclusion

In conclusion, passive learning is as a powerful tool for to study systems behavior from execution traces when explicit specifications are unavailable. By identifying minimal LTL formulas that distinguish between acceptable and unacceptable behaviors, this approach provides clarity and enhances system understanding.

However, while passive learning has a lot of potential, its effectiveness relies on the correctness and reliability of the learned formulas. This is why we have worked on BMC which will check if the learned formula is correct.

That being said, our current implementation is in progress, a lot work is still needed but it could bring great results.

Copying this document

Chapter 7

Bibliography

[1] Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT Press. (pages 4, 5, 6, and 20)

[2] Bordais, B., Neider, D., and Roy, R. (2023). Learning temporal properties is np-hard. (page 3)

[3] et al, A. B. (2003). Bounded model checking. *Adv. Comput.*, 58:117–148.

[4] Fijalkow, N. and Lagarde, G. (2021). The complexity of learning linear temporal formulas from examples. (page 3)

[5] Hsu, T.-H., Bonakdarpour, B., Finkbeiner, B., and Sánchez, C. (2023). Bounded model checking for asynchronous hyperproperties. (pages 3, 11, 12, 14, and 20)

[6] Neider, D. and Gavran, I. (2018). Learning linear temporal properties. (pages 16 and 17)

[7] Pommellet, A., Stan, D., and Scatton, S. (2024). Sat-based learning of computation tree logic.

Contents