Comparison of LTL and CTL

Zhang Ting(tingz@cs.stanford.edu)
Department of Computer Science
Stanford University
December 16, 2000

Abstract

Temporal logic is one type of modal logic for reasoning about temporal properties. In the last twenty years, it became a widely used tool in specifying and verifying properties of concurrent systems. Among many variants of temporal logics proposed so far, the most popular two are linear time temporal logic (LTL) and branching time temporal logic (BTL). In the following survey, we present a comparison of the two logics from several viewpoints.

1 Introduction

Using temporal logic to reason about correctness of concurrent system was first proposed by Amir Pnueli in 1977. From then on, many types of temporal logics were invented. Two of the most popular ones are linear time temporal logic (LTL) and branching time temporal logic (BTL). Both logics include the common temporal operators $[G]p$ ("always p"), $[F]p$ ("sometimes p"), $[X]p$ ("next p"), $p[U]q$ ("p until q"). In fact, in their original form, both logics have the same set of temporal operators (namely $[G]$ and $[F]$) and hence the same formulas. The essential difference between the two lies in the semantics rather than in the theory itself. The different views of "time" lead to different interpretations. In LTL, the time is a linear discrete sequence such that there is only one possible future at each point. By contrast in BTL, each time point has several possible futures. To accommodate this descriptive ability, BTL includes two additional quantification temporal operators $[A]$ ("for all possible futures") and $[E]$ ("for some possible futures"). When interpreted in the context of concurrent systems, LTL formulas specify properties regarding all computations in the system, and hence they are implicitly universally quantified in the top level. BTL, in addition to unconstrained applications of the operator $[A]$, further provides the ability to express existential properties (e.g., potential deadlock in some future [11]).

Ever since the two logics were introduced, the controversy regarding which logic is a more appropriate tool to reason about concurrent systems never stops in compute science community. In the following survey we present a comparison of the two logics from several viewpoints. There are several variants of BTL among which the computation tree logic (CTL) is most widely used in practice. So we choose CTL as representative of BTL in the following comparison. Sometimes we also include its superset CTL* in the discussion. One of our assumption is both logics are propositional, i.e., atomic formulas are propositions with boolean true/false value assigned at evaluation. This simplification is still expressive enough for our reasoning about program correctness in the sense that a proposition corresponds to a description of system state property. Our comparison is organized as follows. A formal syntax of the two logics is given in Section 2. Section 3 continues with their semantics. In Section 4, we show the proof of the incompatibility of their expressiveness.

Section 5 discusses their respective model generators mainly from an automaton-theoretic viewpoint. Section 6 introduces a model checking method for each logic. Section 7 gives a comparison on model checking complexity. A proof of PSPACE-completeness of LTL satisfiability is given. Section 8 discusses modular model checking and how logic structure itself intrinsically induces different approaches.

2 Syntax

2.1 LTL Formula

An LTL formula is a boolean combination of or application of temporal operators $[U]$ ("until") and $[X]$ ("next") to subformulas or atomic propositions. Formally,

- $LTL := [A]SF$ (*implicit*)
- $SF := AP \mid \neg SF \mid SF \wedge SF \mid SF[U]SF \mid [X]SF$

where SF is a state formula and P is an atomic proposition.

Other temporal operators can be defined using [U] and [X]. For example,

- $[F]f := true[U]f$
- $[G]f := \neg[F]\neg f$

where SF and P are as before and PF is a path formula.

2.2 CTL* Formula

CTL* is equipped with two additional path temporal operators [A] ("in all paths") and [E] ("in some path"). Since every formula should be evaluated on some path or on all paths, we have the restriction that every [U] or [X] temporal operator must be quantified by a path temporal operator. To define it formally, we usually distinguish state formula (stating a property of a state) and path formula (stating a property along a path). CTL* formulas are defined to be all such state formulas. It is noted that this classification is somewhat artificial and just for helping understanding. Formally,

- $CTL^* := SF$
- $SF := AP \mid \neg SF \mid SF \vee SF \mid [A]PF \mid [E]PF$
- $PF := SF \mid \neg PF \mid PF \wedge PF \mid [X]PF \mid PF[U]PF$

2.3 CTL Formula

A CTL formula is a CTL* formula with the further restriction that every [U] or [X] operator must be immediately quantified by a path operator (either $[A]$ or $[E]$). Formally,

- $CTL ::= SF$
- $SF ::= AP \mid \neg SF \mid SF \wedge SF \mid [A]PF \mid [E]PF$
- $PF ::= SF[U]SF \mid [X]SF$

Notice that the definition of path formula distinguishes CTL and CTL*. In CTL, a path formula is constructed by applying a temporal operator directly to state formula. As a consequence, every temporal operator is immediately quantified by a path operator [A] or [E]. In contrast, CTL* relaxes this restriction to allow any boolean combination of or nested temporal operations on state formulas. As a consequence, though every temporal operator is still required to be quantified, there is no restriction on quantifier positions. Clearly, CTL is just a fragment of CTL*. So is LTL because of the implicit top-level universal quantification.

Since in CTL every path quantifier is combined with a nonpath temporal operator, we can rewrite its syntax as follows.

- $CTL ::= SF$
- $SF ::= AP \mid \neg SF \mid SF \vee SF \mid SF[AU]SF \mid SF[EU]SF$
- $SF ::= [AX]SF \mid [EX]SF$

$[AU]$, $[AX]$, $[EU]$, $[EX]$ are new temporal operators and (semantically) satisfy,

- $f_1[AU]f_2 = [A](f_1[U]f_2)$
- $f_1[EU]f_2 = [E](f_1[U]f_2)$
- $[AX]f = [A][X]f$
- $[EX]f = [E][X]f$

Similarly as in LTL, we can define other operators as abbreviation of the above four.

- $[AF]f = true[AU]f$
- $[EF]f = true[EU]f$
- $[AG]f = \neg[EF]\neg f$
- $[EG]f = \neg[AF]\neg f$

$[A\nabla]$ is called a universal operator and $[E\nabla]$ is called an existential operator.

Note that it is not necessary to have both universal operators and existential operators. For simplicity, we can dispense with [AU] and [AX] if we introduce [EG] as a primitive. I.e., we define,

- $[AX]f = \neg[EX]\neg f$
- $f_1[AU]f_2 = \neg((\neg f_2[EU](\neg f_1 \wedge \neg f_2)) \vee [EG]\neg f_2)$
- $[EF]f = true[EU]f$

- $[AG]f \;=\; \neg[EF]\neg f$
- $[AF]f \;=\; true[AU]f$

Similarly we can remove [EU] and [EX] with [AG] introduced. I.e., we have,

- $[EX]f \;=\; \neg[AX]\neg f$
- $f_1[EU]f_2 \;=\; \neg((\neg f_2[AU](\neg f_1 \wedge \neg f_2)) \vee [AG]\neg f_2)$
- $[AF]f \;=\; true[AU]f$
- $[EG]f \;=\; \neg[AF]\neg f$
- $[EF]f \;=\; true[EU]f$

We will use the former notation in the discussion of CTL model checking in Section 5 and use the latter one in the proof of incompatible expressiveness of CTL and LTL in Section 4.

3 Semantics (Models)

The general framework for modal logic semantics is a Kripke structure $\langle W, R, V\rangle$ where

(1) W is a finite or enumerable set of possible worlds (or states).

(2) R is a binary relation on W.

(3) V is a function $W \to 2^P$ where P is the set of atomic propositions.

V is a labeling function associating each world with a set of true propositions. By symmetry, f can also be defined as a function $P \to 2^W$.

We assume R to be total in dealing with infinite computations. In practice, finite computations can be viewed infinite by extending the last state forever.

The difference between LTL and CTL/CTL* is the definition of the relation R. In LTL, R is restricted to a function. Hence for every world, there is only a unique world as its successor. By contrast, in CTL/CTL*, every world can have multiple successors corresponding to nondeterministic or concurrent program execution.

An interpretation $\langle M, w\rangle$ is defined as a tuple of structure M and a state w as starting state. For LTL, M is a linearly ordered structure, thus $\langle M, w\rangle$ is called a linear interpretation. While for CTL, M is a directed tree structure and the corresponding $\langle M, w\rangle$ is called a branching interpretation.

Definition 3.1 (linear interpretation). *The truth value of a linear interpretation $\langle M, w\rangle$ satisfies:*

(1) $\langle M, w\rangle \models p$ *iff* $p \in V(w)$

(2) $\langle M, w\rangle \models f_1 \wedge f_2$ *iff* $\langle M, w\rangle \models f_1$ *and* $\langle M, w\rangle \models f_2$

(3) $\langle M, w\rangle \models \neg f$ *iff not* $\langle M, w\rangle \models f$

(4) $\langle M, w\rangle \models [X]f$ *iff* $\langle M, R(w)\rangle \models f$

(5) $\langle M, w\rangle \models f_1[U]f_2$ *iff there exists* $i \geq 0$ *such that* $\langle M, R^i(w) \models f_2$ *and for any* $0 \leq j < i,\ \langle M, R^j(w)\rangle \models f_1$

To define the truth value of a branching interpretation, we need to formalize the notion—"path".

Definition 3.2 (extracted linear interpretation). *A linear interpretation* $\langle M, w\rangle$ *is extracted from a branching interpretation* $\langle M', w'\rangle$ *iff*

- $W \subseteq W'$
- $\forall w \in W:\ f(w) = f'(w)$
- $\forall w \in W:\ (w, R(w)) \in R'$

A branching interpretation is defined in a similar way with rule (4) and (5) replaced for path quantified formula.

Definition 3.3 (branching interpretation). *The truth value of a branching interpretation* $\langle M, w\rangle$ *satisfies:*

(1) $\langle M, w\rangle \models p$ *iff* $p \in V(w)$

(2) $\langle M, w\rangle \models f_1 \wedge f_2$ *iff* $\langle M, w\rangle \models f_1$ *and* $\langle M, w\rangle \models f_2$

(3) $\langle M, w\rangle \models \neg f$ *iff not* $\langle M, w\rangle \models f$

(4) $\langle M, w\rangle \models [A]f$ *iff*
for any linear interpretation $\langle M', w'\rangle$ *extracted from* $\langle M, w\rangle$, $\langle M', w'\rangle \models f$

(5) $\langle M, w\rangle \models [E]f$ *iff*
for some linear interpretation $\langle M', w'\rangle$ *extracted from* $\langle M, w\rangle$, $\langle M', w'\rangle \models f$

Note that for the above branching interpretation to be well-defined, we have to view the extracted linear interpretation as a trace within its embedding branching interpretation.

Definition 3.4 (model). *A true interpretation* $\langle M, w\rangle$ *of formula* f *(either LTL or CTL/CTL*) is called the model of* f*, denoted as* $\langle M, w\rangle \models f$.

Usually we also say a (Kripke) structure M is the model of f, (denoted $M \models f$), if for any state w in structure M. We follow this convention in the following discussion unless specified otherwise.

Note that the (Kripke) structure M in our definition is actually an infinite structure. The model defined in this way is a standard model. In the program correctness literature, since a program is often represented as a finite (Kripke) structure, it is sometimes referred to as a model of formulas (properties). In this context, actually every run (unwinding) of the structure from a starting state, (which then forms an infinite structure) is a standard model. Such definition is sometimes called a nonstandard model. However, the meaning is self-explanalory in context, so we don't bother to distinguish them in later discussions.

4 Expressiveness

Clearly, from the syntax definition, we know LTL and CTL both are a fragment of CTL*. As a consequence, both are less expressive than CTL*. However, the expressiveness between them is incomparable. One direction of this claim is straightforward, since LTL can not express existentiality. To be explicit, there is no formula in LTL capturing the meaning of $[E][F]p$. ([12])

To get the other direction, we need a formal definition of expressiveness.

Definition 4.1 (Model Equivalence). *Formula f_1 and f_2 are model M-equivalent iff $M \models f_1 \iff M \models f_2$. Formula f_1 and f_2 are model equivalent iff they are M-equivalent for any model M.*

Definition 4.2 (Definability). *A set of models (representing a property) is definable in logic A iff they are all models of some formula f in A.*

Definition 4.3 (Expressiveness). *The expressiveness of logic A is equivalent to logic B iff for any formula f in A there is a model equivalent formula g in B and vice versa. I.e., all definable models in one logic are also definable in the other logic.*

We also need to unify the model concepts in CTL and LTL with a Kripke structure as the context.

Definition 4.4 (Model). *([12]) A model M is a pair $\langle W, S\rangle$, where W is a set of states and S is a set of sequences of states and for any $s \in S$ we have $s^+ \in S$.*

For an infinite sequence $s = s_0, s_1, s_2, \ldots$, we have $s^+ = s_1, s_2, \ldots$, i.e., the suffix after the first state. The model in definition 4.4 in fact is a set of linear interpretations of a given structure. For both LTL and CTL, it stands for a set of standard models. However, for CTL the standard models are implicitly constructed. From now on until the end of the section, when we use the word model, we refer to definition 4.4

To show CTL is not more expressive than LTL, we need to find a model definable in LTL but not in CTL. We choose strong fairness (compassion) as the candidate since it is a very important property for communication protocols and its inexpressibility in CTL is often referred as a weak point against CTL.

Theorem 4.1. *Strong fairness property (defined in LTL $[G][F]P \longrightarrow [G][F]Q$) is not definable in CTL.*

Proof. Let $M = \langle W, S\rangle$ be the model for property $[G][F]P \longrightarrow [G][F]Q$. I.e., S consists of all sequences in which if there is an finite number of P-states, there is an infinite number of Q-states. It is equivalent to say that S consists of all sequences in which if there is either finite number of P-states or infinite number of Q-states. Let $M' = \langle W, S'\rangle$ be the model for property $[G][F]\neg P \vee [G][F]Q$. I.e., S' consists of all sequences which have an infinite number of non P-states, or Q-states. Obviously, the two models M and M' are not equivalent since finite P-states imply infinite non P-states but not vice versa. Thus, the two LTL formulas distinguish them. We need to show that no CTL formula can observe this difference. Thus, there is no model equivalent formula g in CTL to express the semantics of $[G][F]P \longrightarrow [G][F]Q$, assuming P and Q are not trivial assertions. The proof is established using induction on the structure of CTL formulas. We only need to focus on the structure with temporal operators involved. The inductive correctness of the boolean combination and atomic propositions follows immediately from the semantics definition.

We adopt the notation of universal temporal operators in Section 2. Intuitively, we need to show that if CTL formula f and g can not distinguish model M from M', then $[AG]f$, $[AX]f$, $f[AU]g$ all three can not either. Formally, we need to establish the following three semantics equations for both M and M'.

$$\langle M, x\rangle \models [AG]f \iff \forall y \in W, \langle M, y\rangle \models f \tag{4.1}$$

$$\langle M, x\rangle \models [AX]f \iff \forall y \in W, \langle M, y\rangle \models f \tag{4.2}$$

$$\begin{aligned}\langle M, x\rangle \models f[AU]g \iff & \underbrace{\langle M, x\rangle \models g}_{(a)} \bigvee \\ & \underbrace{(\langle M, x\rangle \models f \wedge \forall y \in W, (\langle M, y\rangle \models f \vee \langle M, y\rangle \models g)}_{(b)} \wedge \\ & \underbrace{(\forall \neg p \in W, \langle M, \neg p\rangle \models g \wedge \forall q \in W, \langle M, q\rangle \models g))}_{(c)}\end{aligned} \tag{4.3}$$

Equation (4.1) and (4.2) follow from the observation that either M or M' only specifies eventual properties and hence put no restriction on the locality of the state sequence. Thus for any state x and y, there is a sequence of the form $\ldots, x, y, \ldots$ In equation (4.3), we denote $\neg p$ and q as any non P-states and Q-states respectively.

In (4.3), part (a) states the eventuality $\langle M, x\rangle \models f[AU]g$ holds immediately. Part (b) gives a necessary condition if this eventuality has to be propagated to later states. Part (c) states that property g is guaranteed to be satisfied in the future. Part (b) and (c) together form the necessary and sufficient condition for the eventuality to hold if it doesn't in the initial states. Since part (a) and (b) don't make any assumption on underlying models, their truth value is the same for both M and M'. We only need to show why part (c) guarantees g to be satisfied in both models.

($\Longrightarrow$) Note in both M and M', there exist two sequences of the form $x, \neg p, \neg p, \ldots$ and $x, q, q, \ldots$ for any non P-states (denoted as $\neg p$) and any Q-states (denoted as q).

($\Longleftarrow$) Note in both M and M', a sequence either contains infinitely many non P-states or infinite Q-states. Then in both models and any sequences there will be always a non P-state or Q-state after any state x. With part (c) we have in both models g will be true somewhere along all paths starting from state x. □

In summary we established that M is definable in LTL but not in CTL.

Corollary 4.1. *LTL formula $[F][G]P$ is not definable in CTL.*

Proof. Suppose it is definable in CTL, then $[F][G]\neg P$ is also definable in CTL, say as formula X, which means along any path P-states happens finitely often. It is known that along any path Q-states happens infinitely often can be defined in CTL as $[AG][AF]Q$. Then $X \vee [AG][AF]Q$ means along any path if P-states happens infinitely often then Q-states, a contradiction to theorem 4.1. □

5 MODEL GENERATOR (Automaton Theoretic View)

Given a temporal formula (regardless whether LTL or CTL) and its defined semantics, we need a way to characterize all models a formula has. In this section, we adopt the automaton-theoretic viewpoint that a model of a temporal formula can be thought as an infinite word (or tree) on alphabet 2^P and then all models together form a language on 2^P. We show that every such language is exactly accepted by some automaton. We use the term model generator to refer to such an automaton.

5.1 Model Generator of LTL formula

A linear interpretation of a LTL formula is an infinite sequence of states (as we here only consider infinite computations)

$$w_0, w_1, w_2, \ldots$$

whose corresponding proposition labeled sequence is

$$V(w_0), V(w_1), V(w_2), \ldots$$

where each of $V(w_i)$ is a subset of P.

The labeled sequence can be viewed as an infinite word over alphabet 2^P. Thus all models of a formula form a language over alphabet 2^P. It turns out that the language obtained above can be recognized by automata running on infinite words.

Definition 5.1 (Ω-Automaton). *A finite ω-automaton is a tuple defined as $\mathcal{A} = \langle S, \Sigma, S_0, R, Acc\rangle$ with a finite state set S, an input alphabet Σ, initial states set S_0 which is a subset of S, a transition relation R and the acceptance condition Acc. The transition relation is a subset of $S \times \Sigma \times S$. A run r on an infinite word $w = a_0a_1a_2\ldots$ where a_i is a letter in Σ is a sequence $r = b_0b_1b_2\ldots$, such that $b_0 \in S_0$ and $R(b_i, a_i, b_{i+1})$ for all $i \geq 0$. An infinite word is accepted by an ω-automaton iff its run is accepted by the automaton.*

To define a deterministic ω-automaton, the transition relation is restricted to be a function of $S \times \Sigma \to S$ hence a run should satisfy $p_{i+1} = R(p_i, a_i)$.

To define acceptance conditions, we need to have a notation

$inf(r) = \{b \mid b \in S \text{ and for infinitely many } i \geq 0, b_i = b\}$

Intuitively, inf(r) defines a set of states which are "visited" infinitely often by the run r.

The definition 5.1 gives the general framework of ω-automata on infinite words. According to different requirements of acceptance condition, we can have several types of ω-automata. The four most often used are Büchi, Muller, Rabin, and Streett automata. The expressive power of each nondeterministic version is the same ([4], [13]), however, the complexity of operations like determinization (if possible), complement and nonemptiness decisions may be quite different. We shall go through with Büchi automata which is a sufficient description tool for most discussions.

Definition 5.2 (Büchi Acceptance Condition). *In a Büchi automaton the acceptance condition is given by a set of states $F \subseteq S$. A run is accepted if $inf(r) \cap F \neq \emptyset$. An ω-automaton $\mathcal{A} = \langle S, \Sigma, S_0, R, Acc\rangle$ with Büchi acceptance condition is called a Büchi automaton. A language is called Büchi-recognizable iff it is accepted by a Büchi automaton.*

Similarly, we can define acceptance condition and recognizable language concepts for Rabin, Muller and Street automata respectively. ([4])

We here present a way to construct an ω-automaton which exactly accepts all models of a given LTL formula. ([1]) First we introduce three concepts as follows.

Definition 5.3 (Closure). *A closure of a formula f is a set containing all subformulas of f and their negation but with the identification of $\neg\neg g$ with g, denoted as $cl(f)$ ([1], [2]).*

Definition 5.4 (Eventuality Closure). *An eventuality closure of a formula f is a set containing all subformulas of f of the form $f_1[U]f_2$, denoted as $el(f)$.*

Definition 5.5 (Propositional Consistency). *A set of temporal formulas F is propositionally consistent iff for any f_1, f_2,*

(1) $f_1 \wedge f_2 \in F$ *iff* $f_1 \in F$ *and* $f_2 \in F$

(2) $\neg f_1 \in F$ *iff* $f_1 \notin F$

Definition 5.6 (Local Automaton). $\mathcal{A} = \langle \Sigma^l, S^l, R^l, S_0^l, S_f^l \rangle$ *where,*

(1) The alphabet $\Sigma^l = 2^{cl(f)}$

(2) The state set $S^l \subseteq \Sigma^l$

(3) The initial state set S_0^l *consists of all subsets of* $cl(f)$ *containing formula* f.

(4) The final state set $S_f^l = S^l$. *I.e., every state is a final state.*

(5) The transition relation R^l *has to satisfy:*
$s' \in R^l(s,a)$ *where* s, $s' \in S^l$ *and* $a \in 2^{cl(f)}$ *iff*

(a) $s = a$, *and*
(b) For all $[X]f$, $[X]f \in s$ *iff* $f \in s'$, *and*
(c) For all $f_1[U]f_2$, $f_1[U]f_2 \in s$ *iff either* $f_2 \in s$ *or both* $f_1 \in s$ *and* $f_1[U]f_2 \in s'$

Definition 5.7 (Eventuality Automaton). $E = \langle A^e, S^e, R^e, S_0^e, S_f^e \rangle$ *where,*

(1) The alphabet $A^e = 2^{cl(f)}$

(2) The state set $S^e = 2^{el(f)}$

(3) The initial state set $S_0^e = \emptyset$

(4) The final state set $S_f^e = \emptyset$

(5) The transition relation R^e *has to satisfy:*
$s' \in R^e(s,a)$ *where* $s, s' \in S^e$ *and* $a \in 2^{cl(f)}$ *iff*

(a) $s = \emptyset$: *for all* $f_1[U]f_2 \in a$, $f_1[U]f_2 \in s'$ *iff* $\neg f_2 \in a$
(b) $s \neq \emptyset$: *for all* $f_1[U]f_2 \in s$, $f_1[U]f_2 \in s'$ *iff* $\neg f_2 \in a$

Claim 5.1 (Automaton Construction). *Given any LTL formula f, there is an automaton $\mathcal{A}$ which can be constructed as the direct product of the local automaton and the eventuality automaton with letter projection from domain $2^{cl(f)}$ onto 2^P.*

5.2 Comparison Automata and Tableau

We now compare the automaton construction with the tableau approach in [2].

The requirements for transition relation R^l in the local automaton enforce that temporal operators are locally satisfied. More precisely, in definition 5.6, we have,

(1) [By (a)] Every edge is labeled the same way as the starting state, which means that every accepted infinite word is the same as the accepting run. Also this requires each edge must be propositionally consistent.

(2) [By (b)]The propositional consistency and then each letter of a word must be a tableau atom.

(3) [By (c)]The transition relation between atoms is established with the initial states to be all f-atom.

Therefore, in all, the local automaton only accept words which are f-reachable path in the corresponding tableau.

To see what type of restriction the eventuality automaton imposes, let us consider an accepting run r on word w. Note that the final acceptance set of the automaton only has one state, namely the $\emptyset$-state. That means that r has to visit the $\emptyset$-state infinitely times. Now for each eventuality formula $f_1[U]f_2 \in cl(f)$ and a position in r of $\emptyset$-state (denoted as r_i), the next state r_{i+1} has three possibilities:

(1) r stays at the $\emptyset$-state, which means $f_1[U]f_2 \notin r_i$

(2) r moves to another non $\emptyset$-state and it is caused by reading a letter which doesn't contain $f_1[U]f_2$. In this case, obviously, $f_1[U]f_2 \notin r_i$.

(3) r moves to another non $\emptyset$-state and it is caused by reading a letter which contains $f_1[U]f_2$. Then (a) in definition 5.7 requires $f_1[U]f_2$ to be included in the next state and (b) requires the automaton can not come back to $\emptyset$-state until it reads a letter which contains f_2. Since there is infinite visits to $\emptyset$-state, it will happen sooner or later at position $j \geq i$.

For an accepted word w, (1)-(3) says for each of $f_1[U]f_2 \in cl(f)$, either $\neg f_1[U]f_2$ or f_2 will be contained in letters of w infinitely often. In tableau term, w is a fulfilling path.

The combination of the two automata produce the fact that every accepted word forms a f-reachable fulfilling path. Recall the theorem in [2], in short, a fulfilling path induces a model. In the proof, the model $s_0, s_1, \ldots$ is constructed by taking each state s_i which agrees with r_i on the interpretation of the propositions. It is essentially our final projection from $2^{cl(f)}$ to 2^P.

Note that the final automaton is a simple product of the local and eventuality automata. Usually a simple product of two ω-automata is not the same as the intersection of two ω-automata. Instead, the former is more restricted than the latter since it only accepts words that simultaneously visit the final states of both infinitely often. However, in the above construction, it works out because the final state set of the local automaton is the same as the state set. As the local automaton accepts any infinite runs, simultaneous visits coincides with separate visits.

The above shows the strong connection between the two model generators. In fact, a tableau can be viewed as an ω-automaton with the node labeling instead of edge labeling that appear more often in literature.

Note that the size of the automata states set $|S| \leq 2^{|cl(f)|} * 2^{|e(f)|} \leq 2^{3*|f|}$. To summarize, we have

Theorem 5.1 (Vardi, Wolper). *Given an LTL formula f, one can build a Büchi sequential automaton $\mathcal{A} = \langle \Sigma, S, T, S_0, F \rangle$ where $\Sigma = 2^P$ and $|S| \leq 2^{O(|f|)}$, such that $L(\mathcal{A})$ is exactly the set of sequences satisfying the formula f.*

5.3 Model Generator of CTL formula

A model of a CTL formula is a branching computation tree, thus we need extend the automata theory with the ability to represent trees.

Definition 5.8 (K-ary Tree). *A generic infinite k-ary tree is a set $N = \{0, 1 \ldots, k-1\}^*$, i.e., the set of all finite words over alphabet $\{0, 1, 2, \ldots, k-1\}$. An alphabet labeled infinite k-ary tree is a triplet $T = \langle \Sigma, N, L \rangle$ where Σ is an alphabet, N is a node set and L is a labeling function such that for any $n \in \{0, 1, 2, \ldots, k-1\}^*$, we have $L(n) \in \Sigma$.*

A tree is implicitly defined in terms of all its prefixes along paths starting at the root.

Definition 5.9 (Tree Automata). *A k-ary infinite tree automaton is a quintuplet defined as $\mathcal{A} = \langle \Sigma, S, S_0, R, Acc \rangle$, where Σ is finite alphabet, S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, R is transition relation of $S \times \Sigma \times S^k$ and Acc is acceptance condition. For $s \in S$, $a \in \Sigma$, we have $\langle s, a, \langle s_0, s_1, \ldots, s_{k-1} \rangle \rangle \in R$ where $s_i \in S$ with $0 \leq i \leq k-1$. A run of a Σ-labeled tree $\langle \Sigma, N, L \rangle$ is a S-labeled tree $\langle S', N, L' \rangle$ where $S' \subseteq S$, N is the same nodes set and L' satisfies $L'(\epsilon) \in S_0$ and $\langle L'(x), L(x), \langle L'(x_0), L'(x_1), \ldots L'(x_{k-1}) \rangle \rangle \in R$. (We use x_i to denote the i-th child of node x.) A run r is accepted by the automaton iff all its infinite branches when viewed as infinite words are accepted by an acceptance condition.*

As in the definition in infinite word automata, there are four widely used acceptance conditions, namely Büchi, Muller, Rabin and Streett condition. Each of the acceptance conditions corresponds to a concrete type of k-ary infinite tree automata. Restricting the transition relation to a function, we have deterministic k-ary infinite tree automata.

To understand transitions in tree automata, we can imagine that upon reading a labeled node x, the automaton first computes k possible next states, then splits into k copies according to each state, and finally traces down each corresponding branch of node x. Suppose the k children of x are $x_0, x_1, \ldots, x_{k-1}$, then the automaton accepts the x-rooted infinite tree iff all its copies accept the k subtrees respectively.

Theorem 5.2 (Vardi, Wolper). *Given a CTL formula f and a constant k, we can construct a k-ary Büchi infinite tree automaton $\mathcal{A} = \langle 2^P, S, S_0, R, F \rangle$, where $|S| \leq 2^{O(|f|)}$, such that $\mathcal{A}$ accepts exactly all k-ary tree models of f.*

The construction of the automaton uses the same techniques as we showed for LTL formulas. As before, a local automaton checks local consistency. However, we need two types of eventuality automata. One checks existential eventuality on some path and the other checks universal eventuality on all paths. Since all construction bears great similarities as before, we here only point out the differences.

Definition 5.10 (Universal Eventuality Closure). *A universal eventuality closure of a formula f is a set containing all subformulas of f of the form $f_1[AU]f_2$, denoted as $ue(f)$.*

Definition 5.11 (Existential Eventuality Closure). *An existential eventuality closure of a formula f is a set containing all subformulas of f of the form $f_1[EU]f_2$, denoted as $ee(f)$.*

Definition 5.12 (Local Automaton). $\mathcal{A} = \langle \Sigma, S^l, R^l, S_0^l, S_f^l \rangle$ *where the transition relation R^l has to satisfy: $\langle s, a, \langle t_0, t_1, \ldots, t_{k-1} \rangle \in R^l \rangle$ where $s, t_i \in S^l$ and $a \in 2^{cl(f)}$ iff*

(1) For all $[E][X]f$, $[E][X]f \in s$ iff $f \in t_i$ for some $0 \leq i \leq k-1$, and

(2) For all $[A][X]f$, $[E][X]f \in s$ iff $f \in t_i$ for all $0 \leq i \leq k-1$, and

(3) For all $[E]f_1[U]f_2$, $[E]f_1[U]f_2 \in s$ iff either $f_2 \in s$ or both $f_1 \in s$ and $f_1[U]f_2 \in t_i$ for some $0 \leq i \leq k-1$

(4) For all $[A]f_1[U]f_2$, $[E]f_1[U]f_2 \in s$ iff either $f_2 \in s$ or both $f_1 \in s$ and $f_1[U]f_2 \in t_i$ for all $0 \leq i \leq k-1$

Definition 5.13 (Universal Eventuality Automaton).

$\mathcal{U}_\mathcal{E} = \langle \Sigma, S^{u,e}, R^{u,e}, S_0^{u,e}, S_f^{u,e} \rangle$ *where $S^{u,e} = 2^{ue(f)}$, $S_0^{u,e} = \emptyset, S_f^{u,e} = \emptyset$ and the transition relation $R^{u,e}$ has to satisfy: $\langle s, a, t_0, t_1, \ldots, t_{k-1} \rangle \in R^{u,e}$ where $s \in S^{u,e}, t_i \in S^{u,e}$ for all $0 \leq i \leq k-1$ and $a \in 2^{cl(f)}$ iff*

(1) $s = \emptyset$: for all $[A]f_1[U]f_2 \in a$, $[A]f_1[U]f_2 \in t_i$ for all $0 \leq i \leq k-1$ iff $f_2 \notin a$.

(2) $s \neq \emptyset$: for all $[A]f_1[U]f_2 \in s$, $[A]f_1[U]f_2 \in t_i$ for all $0 \leq i \leq k-1$ iff $f_2 \notin a$.

Definition 5.14 (Existential Eventuality Automaton).

$\mathcal{A}_\mathcal{E} = \langle \Sigma, S^{e,e}, R^{e,e}, S_0^{e,e}, S_f^{e,e} \rangle$ *where $S^{e,e} = 2^{ee(f)}), S_0^{e,e} = \emptyset, S_f^{e,e} = \emptyset$ and the transition relation $R^{e,e}$ has to satisfy:*

$$\langle s, a, \langle t_0, t_1, \ldots, t_{k-1} \rangle \rangle \in R^{e,e} \text{ where } s \in S^{e,e}, t_i \in S^{e,e} \text{ for all } 0 \leq i \leq k-1 \text{ and } a \in 2^{cl(f)} \text{ iff}$$

(1) $s = \emptyset$: for all $[E]f_1[U]f_2 \in a$, $[E]f_1[U]f_2 \in t_i$ for some $0 \leq i \leq k-1$ iff $f_2 \notin a$.

(2) $s \neq \emptyset$: for all $[E]f_1[U]f_2 \in s$, $[E]f_1[U]f_2 \in t_i$ for some $0 \leq i \leq k-1$ iff $f_2 \notin a$.

For the existential eventuality automaton, we have

Lemma 5.1. *If a computation tree rooted at s_0 is accepted by an existential eventuality automata, then it is the model of all existential eventuality formulas of the form $[E]f_1[U]f_2$ contained in s_0.*

Proof. Consider an arbitrary existential eventuality formula $[E]f_1[U]f_2$ in s_0. The run on the accepted tree will infinitely pass $\emptyset$-state in every branch. Similarly as in definition 5.14, (2) requires downwards somewhere in the run tree rooted at s_0, there is an $\emptyset$-state s' and along the path from s_0 to s' a letter containing f_2 must have been read. Now if the automaton reads a letter containing $[E]f_1[U]f_2$ again, by (1) it will come to a state which is labeled by $[E]f_1[U]f_2$. Repeat the previous argument, it will come back to an $\emptyset$-state iff a letter containing f_2 is read somewhere in the subtree. Thus such states containing f_2 form an infinite path starting from s. If f_2-state is finite, we know $[E]f_1[U]f_2$-state should be finite too, then we can arbitrarily choose a path after using the deepest node of f_2-state. In this extracted path, we can identify $[E]f_1[U]f_2$ with $f_1[U]f_2$ and $\neg[E]f_1[U]f_2$

with $\neg f_1[U]f_2$. Clearly, it is a fulfilling path for $f_1[U]f_2$ and hence the path is a model of $f_1[U]f_2$. According to the interpretation semantics, a branching structure M is a model for $[E]f_1[U]f_2$ iff there is an extracted linear structure M' which is a model of $f_1[U]f_2$. In summary, definition 5.14 guarantees that any computation tree accepted by a run starting from some state s_0 will be a model for all existential eventuality formulas in s_0. $\square$

Similar arguments apply to the universal eventuality automaton and we have,

Lemma 5.2. *If a computation tree rooted at s_0 is accepted by a universal eventuality automaton, then it is the model of all universal eventuality formulas of the form $[A]f_1[U]f_2$ contained in s_0.*

Combination of all above subautomata needs some tricks. Direct simple product does not work since both accepting state sets of two eventuality automata are not total. It will restrict the product to only accept runs in which each branch simultaneously visits three types of final states. As before, there is no difficulty to incorporate the local automaton and difficulty lies in combination of the two eventuality automata. The idea is to make two copies of the simple product; each copy can be viewed a track in the combined automata. One track $\mathcal{T}_{\mathcal{U}}$ is used to check satisfaction of universal eventuality and the other $\mathcal{T}_{\mathcal{E}}$ is used to check satisfaction of existential eventuality. Suppose a run stays at $\mathcal{T}_{\mathcal{U}}$ now, whenever it passes final state of the universal eventuality automaton $\mathcal{A}_{\mathcal{U}}$, it switches to $\mathcal{T}_{\mathcal{E}}$ and continues running there. It will only come back to $\mathcal{T}_{\mathcal{U}}$ when it passes a final state of the existential eventuality automaton $\mathcal{A}_{\mathcal{E}}$. The final state of this new automaton is a product of the final state set of $\mathcal{A}_{\mathcal{U}}$ and the state set of $\mathcal{A}_{\mathcal{E}}$, or vice versa. This forces any accepting run alternatingly to visit the final states of both eventuality automatan infinitely often. Clearly, the final automaton size is still bounded by $2^{O(|f|)}$.

6 Program Verification

Program verification is to prove whether a program satisfies a specification. Two factors strongly influence the verification techniques, namely how to represent the program and the specification. In this article our discussion assumes programs are represented by transition systems and specifications are represented by temporal formulas.

6.1 Transition System

Definition 6.1 (Transition System). *A transition system is a Kripke structure $\mathcal{T} = \langle S, S_0, R, V\rangle$ where,*

(1) S is a finite set of states,

(2) S_0 is a subset of S as initial states,

(3) R is a transition relation on $S \times S$,

(4) V is a labeling function $S \to 2^P$ to represent each state as a set of propositions which are true in the state.

Quite obviously, a transition system $\mathcal{T}$ can be viewed as a nonstandard model, i.e., every unwinding tree structure of $\mathcal{T}$ is a branching interpretation of CTL/CTL* formulas. Furthermore, each path of the unwinding tree structure is a linear interpretation $\langle M_l, w_l\rangle$ of LTL formulas.

6.2 Verification of LTL formulas

We now introduce a system verification technique using the automata approach. As we observed before, a transition system $\mathcal{T} = \langle S, S_0, R, V\rangle$, when unwound from a starting state q, can be viewed as a tree Kripke structure. Each infinite path in the tree forms a computation denoted as

$$V(s_0), V(s_1), V(s_2), \ldots$$

The sequence can also be viewed as an infinite word over 2^P. Hence $\mathcal{T}$ can be thought as an ω-automaton which accepts all its computations (when they are viewed as words over 2^P). Formally for $\mathcal{T} = \langle S, S_0, R, V\rangle$ we have a corresponding automaton

$$\mathcal{A} = \langle \Sigma, S, S_0, R_t, S_f\rangle \text{ where,}$$

(1) $\Sigma = 2^P$

(2) $(s, a, s') \in R_t$ iff $V(s) = a$ and $(s, s') \in R$

(3) $S_f = S$

(2) says every edge is labeled as its starting state and (3) means every infinite run is accepted. Clearly, this automaton accepts all computations $\mathcal{T}$ generates.

A program $\mathcal{P}$ which satisfies a LTL formula f if every computation of $\mathcal{P}$ is a model of f. Since all models of f forms a language $\mathcal{L}$ of an automaton $\mathcal{L}(\mathcal{A})$ (where $\mathcal{A}$ is constructed from the original transition system $\mathcal{T}$), the verification problem reduces to check whether $\mathcal{L}(\mathcal{A})$ is contained in the language of all models of f. Obviously, the verification ends up to be an automaton containment problem.

Unfortunately, language containment problem of two Büchi automata is PSPACE-complete ([1]). In practice, we construct an automaton $\mathcal{A}'$ accepting models of $\neg f$ and check whether the intersection of $\mathcal{L}(\mathcal{A}')$ and $\mathcal{L}(\mathcal{A})$ is empty. Checking nonemptiness is equivalent to check whether there is a path from starting states to a loop which contains a final state ([13]). The graph reachability problem can be solved in linear time with respect to the number of graph nodes and edges ([14]). We will further this complexity discussion in Section 7.

6.3 Verification of CTL formulas

6.3.1 Automata Approach

As in Section 6.2 , we can construct a deterministic Büchi k-ary tree automaton

$$\mathcal{A} = \langle \Sigma, S, S_0, R_t, S\rangle$$

from the transition system

$$\mathcal{T} = \langle S, S_0, R, V\rangle \text{ where,}$$

(1) $\Sigma = 2^P$,

(2) $\langle s, a, \langle t_0, t_1, \ldots, t_{k-1}\rangle\rangle \in R_t$ iff $a = V(s)$ and for any $s' \in S$, if $R(s, s'), s' = t_i$ for some $0 \leq i \leq k-1$.

Here we assume the given k is large enough than the branching degree of any nodes in $\mathcal{T}$. Then the above tree automaton trivially simulates $\mathcal{T}$ by accepting every unwinding computation tree which is generated from a starting state.

Similarly, all models of a CTL formula form a language recognizable by non-deterministic Büchi tree automata. Again, the verification reduces to an automata containment and further to an automata nonemptiness problem.

6.3.2 μ-Calculus Approach

We introduce a new approach exploiting properties of CTL logic itself. Recall that unwinding a transition system $\mathcal{T}$ from a state s generates an infinite computation tree.

Definition 6.2 (f-state). *A state s in a transition system $\mathcal{T}$ is a f-state where f is a CTL formula, if the computation tree generated by unwinding $\mathcal{T}$ from the state s is a model of the formula f.*

Note that we used the f-state concept in the expressiveness discussion but here f can be any temporal formula, and is not restricted to boolean combination of propositions.

The key observation in this verification technique is:

Proposition 6.1 (State-based Model Checking). *A CTL formula f can be identified with a set of f-states S_f in a given transition system $\mathcal{T} = \langle S, S_0, R, V \rangle$ and $\mathcal{T}$ satisfies f iff $S_0 \subseteq S_f$.*

With this proposition, verification reduces to finding S_f and checking set containment.

It is worth noting that proposition 6.1 not only applies to CTL logic, but also LTL. However, there is a huge difference in computation complexity in finding out S_f. CTL is designed on purpose to ease the computation.

First let us review some conventional notations for set manipulation.

$$R(s) = \{t \mid t \in S \text{ and } \langle s,t \rangle \in R\} \tag{6.1}$$
$$R^{-1}(s) = \{t \mid t \in S \text{ and } \langle t,s \rangle \in R\} \tag{6.2}$$
$$R(S') = \{t \mid t \in S,\ s \in S' \text{ and } \langle s,t \rangle \in R\} \tag{6.3}$$
$$R^{-1}(S') = \{t \mid t \in S,\ s \in S' \text{ and } \langle t,s \rangle \in R\} \tag{6.4}$$

Definition 6.3 (State Set). *A state set $S(f)$ (where f is a CTL formula) in a transition system $\mathcal{T} = \langle S, S_0, R, V \rangle$ is recursively defined as follows:*

$$S(false) = \emptyset \tag{6.5}$$
$$S(true) = S \tag{6.6}$$
$$S(p) = \{s \mid p \in V(s)\} \tag{6.7}$$
$$S(\neg f) = S - S(f) \tag{6.8}$$
$$S(f \wedge g) = S(f) \cap S(g) \tag{6.9}$$
$$S([EX]f) = \{s \mid s \in S \wedge (R(s) \cap S(f)) \neq \emptyset\} \tag{6.10}$$
$$S([EG]f) = S(f) \cap S([EX][EG]f) \tag{6.11}$$
$$S(f[EU]g) = S(g) \cup (S(f) \cap S([EX](f[EU]g))) \tag{6.12}$$

The set manipulation for (6.5)-(6.9) is straightforward. And we have the following equations for (6.10)-(6.12)

$$\begin{aligned} S([EX]f) &= \{s \mid \exists t \ s.t. \ (s,t) \in R \wedge t \in S(f)\} \\ &= R^{-1}(S(f)) \end{aligned} \tag{6.13}$$
$$\begin{aligned} S([EG]f) &= S(f) \cap S([EX][EG]f) \\ &= S(f) \cap R^{-1}(S([EG]f)) \end{aligned} \tag{6.14}$$
$$\begin{aligned} S(f[EU]g) &= S(g) \cup (S(f) \cap S([EX](f[EU]g))) \\ &= S(g) \cup (S(f) \cap R^{-1}(S(f[EU]g))) \end{aligned} \tag{6.15}$$

By (6.14), we know $S([EG]f)$ satisfies a fixpoint Equation (6.16)

$$X = S(f) \cap R^{-1}(X) \tag{6.16}$$

Similarly, $S(f[EU]g)$ satisfies a fixpoint Equation (6.17)

$$X = S(g) \cup (S(f) \cap R^{-1}(X)) \tag{6.17}$$

Note that Equations (6.16) and (6.17) don't have unique solutions. To be explicit, let us consider (6.16). It is obvious that any loop contained in $S(f)$ whose nodes all have in-degree 1 is a solution. To compute $S([EG]f)$, we need the following lemmas.

Lemma 6.1. *Any fixpoint solution of (6.16) is a subset of* $S([EG]f)$.

Proof. Let X be such a solution. First we observe that for any $x \in X$, there exists a $x' \in X$ such that $\langle x, x' \rangle \in R$. Otherwise, $x \notin R^{-1}(X)$ gives $x \notin X$, a contradiction.

Now for any state in X, denoted as x_0, we enumerate its successor sequence $x_0, x_1, x_2, \ldots$. Since all these states are in S(f), when the sequence is viewed as a linear interpretation, it is a model for $[G]f$. As a consequence, the containing branching interpretation is a model for $[EG]f$, i.e. x is an $[EG]f$-state. □

By this lemma, we have a well-defined solution for $S([EG]f)$.

Theorem 6.1 (Greatest Fixpoint Solution). *$S([EG]f)$ is the union of all fixpoints of Equation (6.16). Or in other words, $S([EG]f)$ is the greatest fixpoint of Equation (6.16)*

We have a similar result for computing $S(f[EU]g)$.

Lemma 6.2. *$S(f[EU]g)$ is contained in every fixpoint solution of Equation (6.17).*

Proof. By contradiction.

Let X be any solution of Equation (6.17). Suppose there exists a state $s \in S(f[EU]g)$ but $s \notin X$. Since s is a $f[EU]g$-state, we can extract a linear interpretation as its model, namely $s = s_0, s1, s2, \ldots$. By the semantics of such a model, we should have (i)s_j is a g-state for $j \geq 0$, (ii) all s_i (where $0 \leq i \leq j$) is an f-state. That means there is a path from state s to a state in X since all g-states are contained in X by the Equation (6.17). Let y to be the first state in the path such that its successor y' is in X. The y should not be a g-state by the same reason as above. Then y must be an f-state (i.e., $y \in S(f)$), which is imposed by the model semantics.

Now, we have

$$y' \in X \wedge y \in R^{-1}(y') \Longrightarrow y \in R^{-1}(X)$$

which gives

$$y \in X$$

a contradiction. □

Similarly, $S(f[EU]g)$ is well-defined.

Definition 6.4 (Least Fixpoint Solution). *$S(f[EU]g)$ is the intersection of all fixpoint solutions of Equation (6.17). Or in other words, $S(f[EU]g)$ is the least fixpoint solution of Equation (6.17)*

So far, we essentially defined states set of a CTL formula using propositional μ-calculus. Formally,

$$S([EG]f) = \nu X.(S(f) \cap R^{-1}(X)) \tag{6.18}$$

$$S(f[EU]g) = \mu X.(S(g) \cap (S(f) \cap R^{-1}(X))) \tag{6.19}$$

The computation for the greatest and least fixpoints is given in the following lemma.

Lemma 6.3. *$S([EG]f)$ can be computed by applying iteration*

$$K(X) = S(f) \cap R^{-1}(X)$$

on S. More precisely,

$$\begin{aligned} X_0 &= S \\ X_i &= S(f) \cap R^{-1}(X_{i-1}) \ \textit{where } i > 0 \\ &\vdots \\ \textit{until } X_k &= X_{k-1} \end{aligned}$$

Proof. First we show that iteration function K is monotonic in both senses:

(1) Given two states set X, Y, if $X \subseteq Y$ then $K(X) \subseteq K(Y)$

(2) Given any states set X, and any $i \geq 0$ we have $K^{i+1}(X) \subseteq K^i(X)$

(1) is obvious.

We show (2) by induction. Our target is $X_i \subseteq X_{i+1}$ for all $i \geq 0$.

The base case for X_0 is trivial.

Suppose we have $x \in X_{i+1}$ but $x \notin X_i$. The absence of x in X_i must be caused by "prune" in some iteration. Suppose the prune happened at the iteration computing X_j $(0 < j \leq i)$. I.e. $x \notin X_j$ but $x \in X_{j-1}$. On the other hand its presence in X_{i+1} is caused by an iteration on computing X_{i+1}. A state got "pruned" in an iteration computing X_k iff there exists no state $y \in X_{k-1}$ such that $\langle x, y \rangle \in R$. Thus there exists a state $y \in X_i$ and $\langle x, y \rangle \in R$. Since $y \in X_i$, then $y \in X_{j-1}$, then $x \in X_j$, a contradiction.

By monotonicity (2) and the fact the transition system is finite, we know that iteration function K will converge to a fixpoint denoted Z. We claim Z is the greatest fixpoint. Suppose there is another fixpoint Y. It is obvious that $Z \cup Y$ is a fixpoint too. And by monotonicity (1) $Z \cup Y \subseteq K^i(Z \cup Y) \subseteq K^i(Z) = Z$ for any $i \geq 0$, which gives $Y \subseteq Z$. □

Similarly, we have

Lemma 6.4. *$S(f[EU]g)$ can be computed by applying iteration*

$$K(X) = S(g) \cup (S(f) \cap R^{-1}(X))$$

on $\emptyset$.

Proof. Similar to lemma 6.3. □

Example Now let us compute a fixpoint of a transition system whose state graph is shown in figure 1. This example is modified from [15].

Initially,

$$\begin{aligned}
S &= \{0, 1, 2, 3, 4, 5\} \\
S(p) &= \{0, 1, 3, 4\} \\
S(q) &= \{2, 3\} \\
S(p[EU]q) &= \mu X.(S(q) \cup (S(p) \cap R^{-1}(X)))
\end{aligned}$$

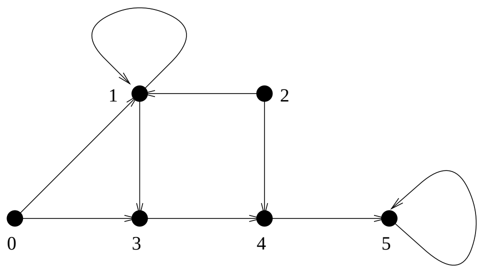

Figure 1: *Transition System Graph*

We have,

$$\begin{aligned} X_0 &= S = \{0,1,2,3,4,5\} \\ R^{-1}(X_0) &= \{0,1,2,3,4,5\} \\ \\ X_1 &= \{2,3\} \cup (\{0,1,3,4\} \cap \emptyset) = \{2,3\} \\ R^{-1}(X_1) &= R^{-1}(\{0,1,2,3,4\}) = \{0,1\} \\ \\ X_2 &= \{2,3\} \cup (\{0,1,3,4\} \cap \{0,1\}) = \{0,1,2,3\} \\ R^{-1}(X_2) &= R^{-1}(\{0,1,2,3\}) = \{0,1,2\} \\ \\ X_3 &= \{2,3\} \cup (\{0,1,3,4\} \cap \{0,1,2\}) = \{0,1,2,3\} = X_2 \end{aligned}$$

Finally,

$$S(p[EU]q) = \{0,1,2,3\}$$

Now we have a well-defined states set and its computation procedure. In summary, we have

Theorem 6.2. *For a transition system $\mathcal{T} = \langle S, S_0, R, V\rangle$ and a CTL formula f, we can find all f-states (denoted as $S(f)$) of $\mathcal{T}$ in a computation bounded by $|S| * |f|$ iterations where $|S|$ is the number of states and $|f|$ is the length of f.*

Proof. We start computing state sets from the inner most subformula of f, and iteratively to the containing subformula until f itself is reached. For subformulas without $[EG]$ $[EU]$ operators, it is pure set manipulation. For each $[EG]$ or $[EU]$ operator involved, we compute the corresponding fixpoint by Kleene iteration. Since each fixpoint computation uses at most $|S|$ iterations and the number of $[EG]$ or $[EU]$ operators is bounded by $|f|$, we conclude that the total number of iterations is bounded by $|S| * |f|$. $\square$

7 Complexity

7.1 Criteria for Measuring Complexity

There are three types of complexity measurement.

- Implementation Complexity

  The complexity is measured against different transition systems with a fixed specification. The complexity with regard to the specification is taken as a constant.

- Specification Complexity

  It is the dual of implementation complexity. We have a fixed transition system and variable specifications.

- Combined Complexity

  It is the total complexity involved when dealing with variable system and specifications.

7.2 Complexity of LTL Model Checking

As we showed in Section 6, we reduce the verification problem to checking nonemptiness of an automaton language. The automaton has states bounded by $O(|S| * 2^{|f|})$ where $|S|$ is the number of states and $|f|$ is the formula length. So the implementation complexity is $O(|S|)$ and the specification complexity is $O(2^{|f|})$.

The implementation complexity could be reduced using symbolic presentation (like OBDD in [16]) for states. However, we can not be more optimistic on the specification complexity. It was proved that determination of truth in a particular finite structure (i.e. transition system) is PSPACE-complete ([17]). Thus there is little room left for improvement unless PSPACE is not equal to EXPTIME.

We show here a weaker conclusion which gives some insights in LTL complexity.

Theorem 7.1. *Satisfiability of LTL is PSPACE-complete*

Proof. Established by the following lemma 7.1 and 7.2. □

Lemma 7.1. *Satisfiability of LTL is in PSPACE.*

Proof. Recall for any LTL formula f, we can construct a Büchi automaton with states size bounded by $2^{|f|}$ which accepts exactly all models of f. The satisfiability problem then reduces to checking non-emptiness of a ω-automaton language, which can be done in linear time with respect to the number of states of the automaton. All we need to show is the space consumed in the automaton construction and checking is polynomial. We resort to a nondeterministic Turning Machine which verifies reachability by first "guessing" a path between two nodes, and then constructing as well as checking the nodes on the path one by one. It always needs to keep only one node in its memory. Since the total number of automaton states of $O(2^{|f|})$ can be coded in $(|f|)$ memory units, it follows that satisfiability of LTL is in NPSPACE. Applying Savitch's theorem ([3]), we proved that LTL is in PSPACE. □

Lemma 7.2. *Satisfiability of LTL is PSPACE-hard.*

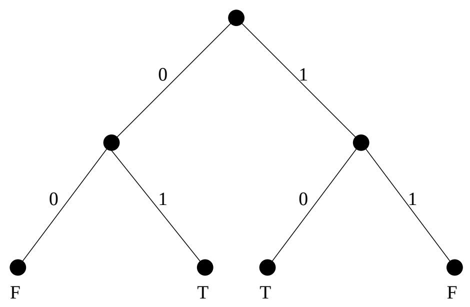

Figure 2: *Game Tree*

Proof. We need to polynomially reduce a PSPACE-hard problem to satisfiability of LTL. Here we choose the TQBF problem. Recall that the TQBF problem is to determine whether a fully quantified Boolean formula is true. Formally,

TQBF = $\{\ f \mid f$ is a fully quantified true Boolean formula.$\}$

For example,

$$f = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

We now need to polynomially reduce TQBF to LTL satisfiability. I.e., we need an algorithm which translates a TQBF formula f_q into a LTL formula f_l in polynomial time and f_q is true iff f_l is satisfiable. The main obstacle is LTL formulas (of our discussion) don't have any quantifier. (Otherwise it would be trivial). Let us think the truth value of a TQBF formula in the viewpoint of game theory. There are two players $\mathcal{A}$ and $\mathcal{E}$ corresponding to universal and existential quantifier respectively. For each move, players only have two choices, namely assign 0 or 1 to a quantified variable. It is much better to identify ourselves as $\mathcal{E}$ and our opponent as $\mathcal{A}$. The common setting in game playing is that we can decide our own move while we can not predict our opponent's. This situation fit well the semantics of existentiality and universality. In fact a TFBF f_q is true iff player $\mathcal{E}$ has a winning strategy. For example, the game tree of formula

$$f_q = \forall x \exists y((x \vee y) \wedge (\neg x \vee \neg y))$$

is shown in figure 2,

The idea of the reduction is to "flatten" the corresponding game tree of f_q into "one dimension space" where a LTL formula f_l is able to encode it. Imagine the game is going on in a time interval. In each moment, upon making a different a moving decision, the game (magically) splits into two separate time intervals and each player (or more precisely his clone) moves there to continue. To be explicit, for player $\mathcal{A}$, if he chooses to assign 0 to x, one of his clones enters the left time interval and plays there later on. By symmetry, if he chooses to assign 1 to x, the other clone enters the right time interval to continue. For the player $\mathcal{E}$, whatever his move is, time is not split. That captures the fact that $\mathcal{E}$ can always make an optimal move, which is exactly what a winning strategy means. Thus the game can be viewed as many subgames playing at different time intervals such that each of them is just a possible combination of move sequences. Hence player $\mathcal{E}$ has a winning strategy iff he wins in all these time intervals.

In reduction, every boolean variable in the given fully quantified boolean formula (FBF) f_q will be translated to a proposition in a LTL formula f_l. We assume f_q is in prenex normal form. A technique trick is to introduce a unique new proposition called a marker for every variable in

f_q to separate each interval. The purpose is to prevent subgames interacting with each other. For example, M_x stands for the marker proposition in separating the x interval. Also we don't allow two seccessive marker.

The construction begins at the outmost part of f_q and iteratively goes on to its subformulas until all quantifiers are eliminated and only boolean combination is left. According to the structure of f_q, it boils down to five cases as follows.

(1) $\forall x.f$

$$\begin{aligned}&true\ \ [U]\ \ (M_x \wedge \neg[X]M_x)\ \ [U]\ \ x\ \ [U]\\ &\qquad\qquad (M_x \wedge \neg[X]M_x)\ \ [U]\ \neg x\ \ [U]\ \ (M_x \wedge \neg[X]M_x)\end{aligned} \tag{7.1}$$

This set two time intervals both of which are bounded by markers M_x. In the left interval x always holds and in the right interval $\neg x$ holds. The illustration follows.

—————— M_x —————— x —————— M_x —————— $\neg x$ —————— M_x ——————

(2) $\exists x.f$

$$\begin{aligned}&(true\ \ [U]\ \ (M_x \wedge \neg[X]M_x)\ \ [U]\ \ x\ \ [U]\ \ (M_x \wedge \neg[X]M_x))\\ &\qquad\qquad\qquad \bigvee\\ &\qquad (true\ \ [U]\ \ (M_x \wedge \neg[X]M_x)\ \ [U]\ \ \neg x\ \ [U]\ \ (M_x \wedge \neg[X]M_x))\end{aligned} \tag{7.2}$$

This set one time interval which are bounded by two markers M_x. In the interval x is either true all times or false all times. The illustration follows.

—————— M_x ———————————— x ———————————— M_x ——————

OR

—————— M_x ———————————— $\neg x$ ———————————— M_x ——————

(3) $\nabla x \forall y.f$ where ∇ is either $\forall$ or $\exists$

$$\begin{aligned}&(M_x \wedge [X]x \rightarrow M_x\ \ [U]\ \ (M_y \wedge \neg[X]M_y \wedge x)\ \ [U]\\ &\qquad (x \wedge y)\ \ [U]\ \ (M_y \wedge \neg[X]M_y \wedge x)\ [U]\\ &\qquad\qquad (x \wedge \neg y)\ \ [U]\ \ (M_y \wedge \neg[X]M_y \wedge x)\ \ [U]\ \ M_x)\\ &\qquad\qquad\qquad \bigwedge\\ &(M_x \wedge [X]\neg x \rightarrow M_x\ \ [U]\ \ (M_y \wedge \neg[X]M_y \wedge \neg x)\ \ [U]\\ &\qquad (\neg x \wedge y)\ [U]\ \ (M_y \wedge \neg[X]M_y \wedge \neg x)\ \ [U]\ \ (\neg x \wedge \neg y)\ \ [U]\\ &\qquad\qquad (M_y \wedge \neg[X]M_y \wedge \neg x)\ \ [U]\ \ M_x)\end{aligned} \tag{7.3}$$

The semantics of (3) is similar to (1). Basically it sets two time intervals both of which are bounded by markers M_y. In the left interval y always holds and in the right interval $\neg y$ holds. The difference is the value of x, as background, never change in this interval. I.e. x will always stay true or false in this interval. The illustration follows.

background $\searrow$

———— M_x ———————— $x \vee \neg x$ ———————— M_x ————

foreground $\searrow$

———— $M_x M_y$ ———— y ———— M_y ———— $\neg y$ ———— $M_y M_x$ ————

(4) $\nabla x \exists y.f$ where ∇ is either $\forall$ or $\exists$

$$
\begin{array}{l}
(M_x \wedge [X]x \rightarrow \\
\quad (M_x \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge x) \;\; [U] \;\; (x \wedge y) \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge x) \;\; [U] \;\; M_x) \\
\quad \vee (M_x \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge x) \;\; [U] \;\; (x \wedge \neg y) \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge x) \;\; [U] \;\; M_x)) \\
\\
\qquad\qquad\qquad\qquad\qquad\qquad \bigwedge \\
\\
(M_x \wedge [X]\neg x \rightarrow \\
\quad (M_x \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge \neg x) \;\; [U] \;\; (x \wedge y) \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge \neg x) \;\; [U] \;\; M_x) \\
\quad \vee (M_x \;\; [U] \;\; (M_y \wedge \neg[X]M_y \wedge \neg x) \;\; [U] \;\; (\neg x \wedge \neg y) \;\; [U] \;\; (M_y \wedge [X]M_y \wedge \neg x) \;\; [U] \;\; M_x))
\end{array}
\tag{7.4}
$$

The semantics of (4) is similar to (2). Basically it sets one time intervals which is bounded by two markers M_y. In the interval y is either true all times or false all times. The difference from (2) is x never changes value too. For example,

background $\searrow$

———— M_x ———————— $x \vee \neg x$ ———————— M_x ————

foreground $\searrow$

———— $M_x M_y$ ———————— y ———————— $M_y M_x$ ————

OR

———— $M_x M_y$ ———————— $\neg y$ ———————— $M_y M_x$ ————

Note that in all constructions (1)-(4), the truth value of ordinary propositions at any mark position is irrelevant.

(5) $\mathcal{B}^+$ Boolean combinations

$$[G]\mathcal{B}^+ \tag{7.5}$$

This basically requires $\mathcal{B}^+$ holds in every time point, hence in every time interval.

The final LTL formula f_l is the conjunction of all subformulas generated by applying step (1)-(5) iteratively.

Note that the translation procedure is local in the sense that it consumes a quantifier and quantified variable at each processing step. In each iteration, we only need to know the previous quantified variable. Obviously this translation can be implemented in polynomial time.

By now we finished the polynomial reduction from TFBF to LTL satisfiability. Since the former is PSPACE-hard, it follows that LTL is PSPACE-hard too. $\square$

Note that although the specification complexity of system verification using LTL is PSPACE-complete ([17]), the above theorem doesn't establish it. Since we only can reduce determination of truth in particular structure (e.g. transition system) to satisfiability problem, but not vice versa.

7.3 Complexity for CTL Model Checking

The automata approach for CTL model checking has the same complexity as the LTL counterpart. I.e., we have $O(|S| * 2^{|f|})$. Note that the nonemptiness check for Büchi infinite tree automata is quadratic in $|V| + |E|$ (as reported in [1]). However, the complexity of the specification is still bounded by $2^{|f|}$.

It is a quite surprising result that the specification complexity of CTL model checking is in P while the counterpart in LTL is PSPACE-complete. In technique viewpoint, CTL corresponds to μ-calculus with only one degree alternation. In labeling states with the formula to be proved, the fixpoint computation begins at the inner most subformula and each time proceeds from a subformula to its closest enclosing subformula. The whole procedure involves no recursions. LTL can not enjoy the same advantage since its alternation degree could be linear to the length of the formula. From an intuitive viewpoint, as is observed in [18], CTL is state-based, hence labeling states with formulas involves no further unwinding of the Kripke structure which represents the transition system. In contrast, LTL is path based and since each state can appear in many paths, the labeling can not avoid splitting states to accommodate needs of different paths. This complexity difference is often used by some researchers in arguments advocating CTL.

8 Modular Model Checking

When the model checking complexity is evaluated, the implementation complexity should play the more important role than specification complexity, as was argued by Pnueli et al. Since a practical system usually consists of many parallel components, the total number of states of the system would be the product of all its submodules. It will exponentially quickly build up states such that even a linear implementation complexity may not avoid this state explosion problem. One promising approach is modular model checking. It not only deals with complexity but also provides a good programming methodology beneficial to program synthesis.

As introduced in [19], a general proof framework could be,

$$\frac{M_1 \models f_1 \ , \ M_2 \models f_2 \ , \ C(f_1, f_2, f)}{M_1 \parallel M_2 \models f} \tag{8.1}$$

It means when module M_1 satisfies f_1 and module M_2 satisfies f_2, f (which is logically related to f_1 and f_2 by the connection formula $C(f_1, f_2, f)$) will be satisfied by $M_1 \parallel M_2$ (which is the parallel composition of M_1 and M_2). One of typical connection formula could be $f_1 \wedge f_2 \longrightarrow f$. Thus the verification of a large system can be decomposed to verifications of its modules, which usually involves much smaller state spaces.

To understand the above proof framework, we need to formalize parallel composition concept which should capture the intuition that a composed module contains less "behavior" than each of its submodules and modules are comparable with respect to "behavior complexity". ([9])

Definition 8.1 (structure simulation). *Let M and M' be two structures (or transition systems) where*

(1) $M = \langle S, S_0, R, V, AP\rangle$ and $M' = \langle S', S'_0, R', V', AP'\rangle$ are defined as before with the exception of distinguishing AP and AP' (atomic proposition sets).

(2) $AP' \subseteq AP$

(3) For state $t \in S$ and state $t' \in S'$, a relation $H \subseteq S \times S'$ is a simulation from (M, t) to (M', t') iff the following condition holds:

- *(a) $H(t, t')$;*
- *(b) For all s and s', $H(s, s')$ implies*
  - *(i) $V(s) \cap AP' = V'(s')$; and*
  - *(ii) for every fair path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M there exists a fair path $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' such that $H(s_i, s'_i)$ for all $i \geq 0$.*

We say H is a simulation from M to M' iff for every $s \in S$ there exists $s' \in S'$ such that $H(s, s')$ (I.e. H simulates from (M, s) to (M', s')). In such case, we call M the simulating module and M' the simulated module.

The simulation from M to M' defined above is in fact a homomorphism from M to M'. Thus M' can be viewed as M's abstraction containing an embedded isomorphic image of M.

Definition 8.2 (simulation preorder). *For $s \in S$ and $s' \in S'$, $(M, s) \preceq (M', s')$ iff there is a simulation from (M, s) to (M', s'). $M \preceq M'$ iff there exists a simulation from M to M'*

This definition establishes a preorder (as shown below). Intuitively, a module M is "less than" another module M' iff M contains less behaviors and is more concrete than M'.

Definition 8.3 (structure composition). *Let M and M' be two structures (or transition systems). The composition of M and M', denoted as $M \parallel M'$ is a structure M'' satisfying,*

(1) $S'' = \{(s, s') \mid V(s) \cap AP' = V'(s') \cap AP\}$

(2) $S''_0 = (S_0 \times S'_0) \cap S''$

(3) $AP'' = AP \cup AP'$

(4) $V''((s, s')) = V(s) \cup V'(s')$

(5) $R''((s, s'), (t, t'))$ iff $R(s, t)$ and $R'(s', t')$

The composed system defined above can be viewed as the product of M and M' with conflicting states pruned. Here we omit the fairness condition. Though it is important for a meaningful composition, it doesn't affect the following discussion.

With all these concepts introduced, O. Grumberg et al. showed the following important properties of composition and the simulation relation. ([8]).

Lemma 8.1. *Let $M'' = M \parallel M'$, the following is equivalent*

(1) $\pi'' = (s_0, s'_0)(s_1, s'_1)\ldots$ is fair path in M''

(2) $\pi = s_0 s_1 \ldots$ and $\pi' = s'_0 s'_1 \ldots$ are fair paths in M and M' respectively, and (s_i, s'_i) is a state of M'' for all $i \geq 0$

Proof. It follows from the fact that any computation of M'', when projected to any of its submodules, is a computation in that submodule too. □

Note that the above argument doesn't hold for a computation tree since a computation tree in composed module M'' would be restricted to have less branches than any of its submodules. We later show how importantly it affects modular model checking methods in different logics.

Theorem 8.1. *[simulation relation property and composition semantics]*

(1) Simulation relation $\preceq$ is preorder

(2) For all M and M', $M \parallel M' \preceq M$

(3) For all M, M' and M'', if $M \preceq M'$ then $M \parallel M'' \preceq M' \parallel M''$

(4) For all $M, M \preceq M \parallel M$

Proof. Done by definition and set manipulation. □

Theorem 8.2. *[simulation semantics] For any $ACTL^*$ formula f (with atomic propositions in AP') and $M \preceq M'$, we have $M' \models f \Longrightarrow M \models f$.*

Proof. Done by induction on formula f structure. □

This theorem gives the sufficient condition for property preservation in module composition. Note that the restriction to universal quantified logic is not required for the truth of the theorem. Rather, it is motivated by avoiding complexity consideration.

Having all this background, we introduce module composition logic which was first proposed by Amir Pnueli ([19]).

Definition 8.4 (Model Inference). $\langle f \rangle M \langle g \rangle$ *that means whenever a system containing module* M *satisfies a property* f*, the system will also satisfy property* g*.*

Obviously, it requires that $M \parallel M' \models f \Longrightarrow M \parallel M' \models g$ for any module M'.

One of typical inference rule is,

$$\frac{\langle f_1 \rangle M_1 \langle f_2 \rangle \ , \ \langle true \rangle M_2 \langle f_1 \rangle}{\langle true \rangle M_1 \parallel M_2 \ \langle f_1 \wedge f_2 \rangle} \tag{8.2}$$

This inference procedure has a good property in that we can iteratively use the inferred result in previous phase to compose a new module. The verification problem then reduces to prove for every module M an interesting property f_1 and $f_2, \langle f_1 \rangle M_1 \langle f_2 \rangle$ holds. Of course, we need to first decompose a system into submodules and a system specification into subspecifications in such a way that above (or other inference rules) apply. However this step is highly non-trivial and we are not going to discuss it here.

As observed above, the specification for each module should include two parts. One is describing the assumed property of the environment the module is interacting with and the other is describing the guaranteed behavior the module would have present given the assumed environment. These two are usually called **assume-guarantee pair**.

Proposition 8.1. *For LTL formula* f_1 *and* f_2*,* $M \models f_1 \rightarrow f_2 \Longrightarrow \langle f_1 \rangle M \langle f_2 \rangle$

Proof. Suppose we have a system containing M and satisfying f_1. I.e., every computation in the system $\mathcal{T}$ satisfies f_1. Since every such computation can be viewed as a computation of M (by module composition property in lemma 8.1), it must also satisfy f_2 by the fact $M \models f_1 \rightarrow f_2$. $\square$

By this proposition, the system verification using LTL coincides with verifying each subsystems using the same logic. Therefore the same techniques still apply.

However, it is not the case for CTL module checking. We lose the ability of saying "every computation" in the proof of proposition 8.1. Again as was explained in the comments for lemma 8.1, the simulation relation is defined on paths. It guarantees a simulating path embedded in the simulated module, however, it is not necessarily true for a computation tree. Counterexamples can be imagined as a simulating module which only contains computation trees with some branches pruned off.

To find assume-guarantee pair, we need the following concept ([10]).

Definition 8.5. *[Maximal Model] A module* M *is a maximal model of a formula* f *if* $M \models f$ *and for any module* M'*,* $M' \preceq M \Longrightarrow M' \models f$*.*

Intuitively, M is a maximal model for f if it allows all behavior consistent with f. Formally,

Theorem 8.3. *[Property Preservation in Composition] For a module* M *and formula* f *and* g*, we have* $\langle f \rangle M \langle g \rangle$ *iff* $M \parallel M_{max} \models g$ *where* M_{max} *denotes the maximal model of* f*.*

Proof. $(\Longleftarrow)$

$$
\begin{aligned}
&M \parallel M' \models f && \text{by assumption} && (8.3)\\
&M_{max} \models f && \text{by definition 8.5} && (8.4)\\
&M \parallel M' \preceq M_{max} && \text{by definition 8.5} && (8.5)\\
&M \parallel M \parallel M' \preceq M \parallel M_{max} && \text{by theorem 8.1 and (8.5)} && (8.6)\\
&M \preceq M \parallel M && \text{by theorem 8.1} && (8.7)\\
&M \parallel M' \preceq M \parallel M \parallel M' && \text{by theorem 8.1 and (8.7)} && (8.8)\\
&M \parallel M' \preceq M \parallel M_{max} && \text{by theorem 8.1, (8.6) and (8.8)} && (8.9)\\
&M \parallel M_{max} \models g && \text{by the given antecedent} && (8.10)\\
&M \parallel M' \models g && \text{by theorem 8.2} && (8.11)
\end{aligned}
$$

($\Longrightarrow$)

$$
\begin{aligned}
&M \parallel M_{max} \preceq M_{max} && \text{by theorem 8.1} && (8.12)\\
&M \parallel M_{max} \models f && \text{by theorem 8.2} && (8.13)\\
&M \parallel M_{max} \models g && \text{by antecedent and (8.13)} && (8.14)
\end{aligned}
$$

□

Having theorem 8.3, the modular model checking for CTL formula reduces to finding a maximal model of a formula f. A maximal model of a formula f can be constructed essentially by taking the product of a general transition system (whose transition graph is a complete graph) and an automaton which exactly accepts the computation tree satisfying f. Note that to establish the assume-guarantee pair, CTL modular model checking has to find a maximal model which suffers exponential complexity. In summary, we have the following complexity result for CTL modular model checking (reported in [11]).

Theorem 8.4 (Modular Checking Complexity [11]).

(1) The branching modular model-checking problem for ACTL is PSPACE-complete.

(2) The branching modular model-checking problem for $ACTL^*$ is EXPSPACE-complete.

(3) The branching modular model-checking problem for CTL is EXPTIME-complete.

9 Concluding Remarks

There are extensive comparisons between LTL and CTL in the program verification literature. A well-known quote is "branching is free". It is based on the dominance of CTL over LTL in specification complexity and the fact that many CTL-based model checking products are widely used in computer hardware industry. However, it is too simplistic to have a clear-cut conclusion. LTL has its strong points in the following areas.

9.1 Expressiveness

Although the lack of existential quantification in LTL is considered a big drawback of its expressiveness, usually, we think LTL is more expressive in the sense that lots of system properties concerned can be expressed easily in LTL. For example, an important fairness requirement—compassion (strong fairness) can not be expressed in CTL while there is very natural representation in LTL, namely $[G][F]p \longrightarrow [G][F]q$. Quite naturally, when we think of a property of a system, usually, we think of something holding in all computation paths rather than in some computation trees. We may say the expressiveness of LTL "coincides" with human intuition.

9.2 Modular Model Checking

Although specification complexity is important, many researchers argue that in model checking, the implementation complexity (program complexity) is more important and would be the bottle neck. It is quite true since for a practical system, the number of system states can be tremendously huge, which is often referred as state explosion problem. Even linear complexity can make a system suffer state-explosion. One possible way to handle it is modular model checking. However, in modular model checking, LTL and ACTL has the same PSPACE complexity. The penalty to find the maximal model for an ACTL formula compensates for LTL. One important observation in modular model checking is that to satisfy assume-guarantee condition, we may use strong assumptions even for very weak guarantee. However, the strong assumption may not be expressible in ACTL, which as a weak point of expressiveness, might result in even higher complexity. (If EXPTIME and PSPACE are not equal.)

9.3 Deductive Approach

It is noted that the complexity drawback of LTL is set in the context of the algorithmic verification approach (model checking). However, algorithmic approach has its application limit. It usually requires finite state systems, limiting its application mostly to hardware systems while the deductive approach can handle infinite state systems and parameterized finite-state systems of arbitrary size. The deductive approach in LTL has been extensively studied and practical products (like STEP) are available while deduction system for CTL seldom appears in literature. The first deductive system for proving branching temporal property was designed by Limor Fix ([6]) in 1995, however, there is no practical product up to now. Partial reason may be due to the unnatural semantics of CTL. As was observed by Lamport in [12], LTL can easily formalize many proof techniques introduced by Owicki in reasoning about parallel programs ([7]). However, even if the same proofs can apply in CTL, it will not be as simple and natural as using LTL.

In Summary, the "branching" is not that free. In fact, both CTL and LTL have their attractive points. It would be a promising while challenging research direction to find new logics or techniques which can combine both merits.

Acknowledgments

The authors thank to Tomás E. Uribe for his kindness and valuable comments on infinite tree-automaton.

References

[1] P. Wolper (University de Liege, Belgium) **On the Relation of Programs and Computations to Models of Temporal Logic**

[2] Zohar Manna and Amir Pnueli. Springer 1995. **Temporal Verification of Reactive Systems—Safety.**

[3] Michael Sipser. PWS Publishing Company. **Introduction to the theory of computation**

[4] Wolfgang Thomas. May 1996. **Languages, Automata, and Logic.**

[5] Sergey Berezin, Edmund Clarke, Somesh Jha, Will Marrero. September 23, 1996. **Model Checking Algorithms for the mu-Calculus.**

[6] Limor Fix. September 21, 1995. **Verification of Temporal Properties.**

[7] S. Owicki and D. Gries. Ph.D. Thesis, Cornell University 1975. **Axiomatic Proof Techniques for Parallel Programs.**

[8] Moshe Y. Vardi. **Linear vs. Branching Time: A Complexity-Theoretic Perspective.**

[9] Orna Grumberg, David E. Long August 15, 1991. **Model Checking and Modular Verification.**

[10] Orna Kupferman and Moshe Y. Vardi **Modular Model Checking.**

[11] E. Allen Emerson and Chin-Laung Lei. Proceedings of ACM of programming language 1985 pp84-96. **Modalities for Model Checking: Branching Time Striking Back.**

[12] Leslie Lamport. Proceedings of ACM of programming language 1980 pp174-185. **"Sometimes" is sometimes "not never" on the temporal logic of programs.**

[13] Moshe Y. Vardi. **An automata-Theoretic Approach to Linear Temporal Logic.**

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Riverst. The MIT Press 1990. **Introduction to Algorithms.**

[15] G.L.J.M. Janssen, TABLEAUX'98 Tutorial. **Implementation of Propositional Temporal Logics using BDDs.**

[16] Kenneth L. McMillan Ph.D. thesis **Symbolic Model Checking.**

[17] A. P. Sistla and E. M. Clarke. Journal of ACM Vol. 32, No. 3, July 1985, pp. 733-749. **The complexity of Propositional Linear Temporal Logics.**

[18] Yonit Kesten, Amir Pnueli, Li-on Raviv. November 16, 1998. **Algorithmic Verification of Linear Temporal Logic Specifications.**

[19] Amir Pnueli. Logics and Models of Concurrent Systems. Series F: Computer and System Sciences, Vol. pp 123-144. **In Transition From Global to Modular Temporal Reasoning about Programs.**