Temporal Logic

written report for the seminar
"Model Checking"

Wintersemester 2002/2003

FR 6.2 Informatik - Lehrstuhl Prof. Paul

Anton Myagotin
timur@wjpserver.cs.uni-sb.de

November 30, 2002

Introduction

In this discussion we will be concerned with *concurrent systems*, i.e. systems which consist of a set of components that execute together and have some means of communication between the components. Such systems are given by the text of a program or by the diagram for a circuit. There are many types of concurrent systems: asynchronous and synchronous circuits, programs with shared variables, programs that communicate by message passing. They cannot be modelled by their input-output behavior.

In section 1 we define the notion of a Kripke structure, an unifying formalism that represents a concurrent system of any type. In section 2 we classify the different types of temporal logics. In section 3 we introduce the syntax and the semantics of TL. In section 4 we consider sublogics of CTL*: CTL and LTL. Further, in section 5 we compare their expressive powers. In section 6 we take look at the issue of fairness.

1 Kripke Structure

We use a state transition graph called a *Kripke structure* to capture the behavior of concurrent systems. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of *properties* or *atomic propositions*, atomic facts about the state are true in that state. Paths in the Kripke structure model the computations of the system.

Definition 1 *Let AP be a set of atomic propositions. A Kripke structure M over AP is a four-tuple $M = (S, S_0, R, L)$ where*

- *S is a set of states.*
- *$S_0 \subseteq S$ is a set of initial states.*
- *$R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.*
- *$L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

A *path* in the structure M starting from state s is an infinite sequence of states $\pi = < s_0, s_1, s_2, \ldots >$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

variables: $x, y \in \{0, 1\}$
initialization:

$x \leftarrow 1$
$y \leftarrow 1$

start

while true

$x \leftarrow (x + y) \bmod 2$

end of while

end

Figure 1: Listing of sample program

To illustrate the above definition we consider the simple program shown in figure 1. The program has two variables x and y which are initialized to one. In every iteration the program recalculates the value of x as $x = (x+y) \bmod 2$, and the value of y variable stays the same.

We define the Kripke structure $M = (S, S_0, R, L)$ modelling the behaviour of the program as follows:

- The set of states S is the set of all boolean pairs $(x,\ y)$ representing the values of program variables:
  $S = \{s_0, s_1, s_2, s_3\} = \{(1, 1), (1, 0), (0, 1), (0, 0)\}$
- The set of initial states is $S_0 = \{s_0\}$, since the program starts with $x = 1$ and $y = 1$.
- The transition relation derived from the program is $R = \{(s_0, s_2),\ (s_2, s_0),\ (s_1, s_1),\ (s_3, s_3)\}$
- We define the set of atomic proposition as $AP = \{x\_eq\_0, x\_eq\_1, y\_eq\_0, y\_eq\_1\}$ and label each state as follows:
  - $L(s_0) = \{x\_eq\_1, y\_eq\_1\}$
  - $L(s_1) = \{x\_eq\_1, y\_eq\_0\}$
  - $L(s_2) = \{x\_eq\_0, y\_eq\_1\}$
  - $L(s_3) = \{x\_eq\_0, y\_eq\_0\}$

The only path in the Kripke structure that starts in the initial state s_0 is $\pi =< s_0, s_2, s_0, s_2, \ldots >$. (This path is the only computation of the system.) We can draw a state transition graph representing the Kripke structure M as shown in figure 2.

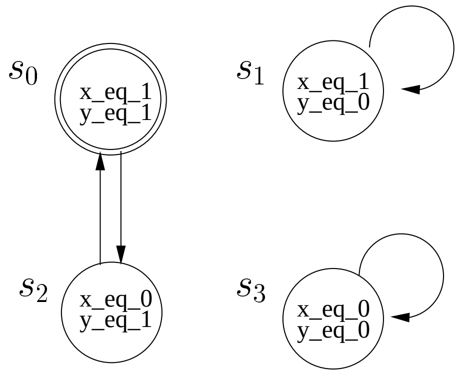


Figure 2: State transition graph of the Kripke structure M

2 Classification of Temporal Logics

A *temporal logic* is a formalism which is used to reason about properties of the computations of a Kripke structure. In all temporal logics that we will consider time is not mentioned explicitly; instead, a formula might specify that *eventually* some state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special temporal operators. These operators can also be combined with boolean connectives or arbitrarily nested.

We can classify most temporal logics into the following cases:

- **Propositional versus first-order**
  In propositional temporal logic the non-temporal portion of the logic is just classical propositional logic. Thus, formulas are built up from atomic propositions, boolean connectives (such as $\neg$, $\vee$, $\wedge$) and the temporal operators. The first-order temporal logic additionally uses functions, predicates, constants, variables, and quantifiers.

- **Branching versus linear time**
  In temporal logics, there are two possible views regarding the nature of time. One is that time is linear: at each moment there is only one possible future moment. The other is that time has a branching, tree-like nature: at each moment, time may split into alternate courses representing different possible futures.

- **Past versus future**
  In most temporal logics only future-tense operators are provided. This appears reasonable since, as a rule, any program computations have a starting time, and it can be shown, as a consequence, inclusion of

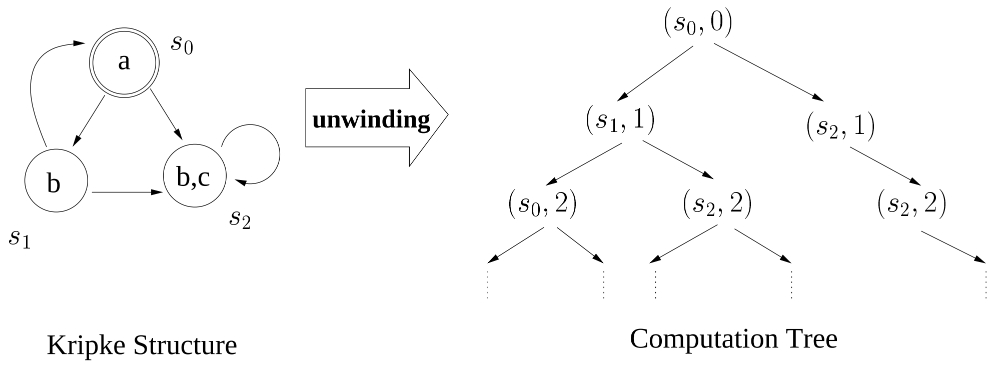


Figure 3: Unwinding Kripke structure into an computation tree.

past-tense operators adds no expressive power. However, past-tense operators are useful for specifying of properties related to the past.

3 CTL*

In this section we will define the computation tree logic CTL* (speak: "CTL star"). The paths starting from some state in a Kripke structure form the so-called *computation tree*. The tree is obtained by designating an initial state in a Kripke structure and then unwinding the structure into infinite tree with the designated state as the root, as illustrated in figure 3. More formally:

Definition 2 *We define the structure $\hat{M} = (\hat{S}, \hat{R}, \hat{L})$, which is called the computation tree. It is obtained by unwinding Kripke structure $M = (S, R, \{s_0\}, L)$ starting at state $s_0 \in S$, where $\hat{S}$ and $\hat{R}$ are respectively the least subsets of $S \times \mathbb{N}$ and $\hat{S} \times \hat{S}$ such that*

- $(s_0, 0) \in \hat{S}$;
- *If* $(s, n) \in \hat{S}$, *then* $\{(p, n+1) : (s, p) \in R\} \subseteq \hat{S}$ *and* $\{((s, n), (p, n+1)) : (s, p) \in R\} \subseteq \hat{R}$;
- $\hat{L}((s, n)) = L(s)$.

The graph of $\hat{M}$ is a tree with the root $(s_0, 0)$. For every path in the Kripke structure M there is a path in computation tree $\hat{M}$ and vice versa. The computation tree shows all of the possible computations starting from the initial state.

The CTL* formulas contain *path quantifiers* and *temporal operators*. The path quantifiers are used in a particular state to specify that all of paths or some of the paths starting at the state have some property. Thus, there are two such quantifiers:

- **A**("for all paths starting from a state ...")
- **E**("for some path starting from a state ...")

The temporal operators describe properties of the paths through the tree. There are four temporal operators:

- **X**("*next time*") requires that a property holds in the second state of the path.
- The **F**("*eventually*" or "*in the future*") operator is used to assert that property will hold at some state on the path.
- **G** ("*always*" or "*globally*") specifies that a property holds at every state on the path.
- The **U** ("*until*") is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

Syntax

Now, we give a precise definition of the syntax of CTL*. There are two types of formulas in CTL*: *state formulas* (which are true in specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic propositions. The syntax of state formulas is given by the following rules:

(S1) if $p \in AP$, then p is a state formula.

(S2) if f, g are state formulas then so are $\neg f$ and $f \vee g$.

(S3) if f is a path formula then $\mathbf{E}f$ and $\mathbf{A}f$ are state formulas.

Three additional rules are needed to specify the syntax of path formulas:

(P1) Each state formula is also a path formula

(P2) If f, g are path formulas then so are $\neg f$ and $f \vee g$.

(P3) If f, g are path formulas then so are $\mathbf{X}f$ and $f\mathbf{U}g$.

The set of state formulas generated by the above rules forms the language of CTL*.

Semantics

We define the semantics of CTL* with respect to a Kripke structure M. Let $\pi =< s_0, s_1, \ldots >$ be a path in M. We use π^i to denote the suffix of π starting at state s_i. If f is a state formula, the notation $M, s \models f$ means that f holds at the state s in the Kripke structure M. Similarly, if f is a path formula, then $M, \pi \models f$ means that f holds along the path π in the Kripke structure M. When the Kripke structure is clear from the context we will omit it. For state formulas f_1, f_2 and path formulas g_1, g_2 we define inductively the relation $\models$ as follows:

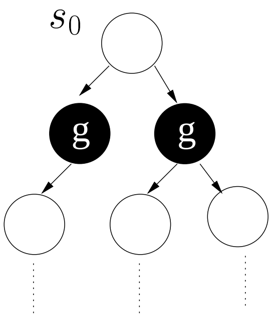
(S1)	$s \models f_1$	$\Leftrightarrow$	$f_1 \in L(s)$, where $f_1 \in AP$
(S2)	$s \models \neg f_1$	$\Leftrightarrow$	if not $s \models f_1$
	$s \models f_1 \vee f_2$	$\Leftrightarrow$	$s \models f_1$ or $s \models f_2$
(S3)	$s \models \mathbf{E} g_1$	$\Leftrightarrow$	$\exists \pi$ starting from s: $\pi \models g_1$
	$s \models \mathbf{A} g_1$	$\Leftrightarrow$	$\forall \pi$ starting from s: $\pi \models g_1$
(P1)	$\pi \models f_1$	$\Leftrightarrow$	$s \models f_1$ and s is the first state of the path π
(P2)	$\pi \models \neg g_1$	$\Leftrightarrow$	if not $\pi \models g_1$
	$\pi \models g_1 \vee g_2$	$\Leftrightarrow$	$\pi \models g_1$ or $\pi \models g_2$
(P3)	$\pi \models \mathbf{X} g_1$	$\Leftrightarrow$	$\pi^1 \models g_1$
	$\pi \models g_1 \mathbf{U} g_2$	$\Leftrightarrow$	$\exists k \geq 0 : \pi^k \models g_2$ and $\forall 0 \leq j < k : \pi^j \models g_1$

Observation 1 *Using the operators* $\neg, \vee, \mathbf{U}$ *and* $\mathbf{E}$ *we can express the other operators* $\wedge, \mathbf{F}, \mathbf{G}, \mathbf{A}$.

- $\mathbf{true} \equiv a \vee \neg a$*, for an arbitrary* $a \in AP$
- $f \wedge g \equiv \neg(f \vee g)$
- $\mathbf{F} f \equiv \mathbf{true}\ \mathbf{U} f$
- $\mathbf{G} f \equiv \neg \mathbf{F} \neg f$
- $\mathbf{A} f \equiv \neg \mathbf{E} \neg f$

4 CTL and LTL

In this section we consider two useful sublogics of CTL*: one is the *computation tree logic* CTL and the other is the *linear time logic* LTL. In CTL the temporal operators quantify over the paths that are possible from a given state. In LTL operators are provided for describing properties along a single



$\mathbf{AX}g$ true iff for all paths $\pi = s_0, s_1, \ldots$ in state s_1
g holds, formally:

(S3)	$s \models \mathbf{AX}g$ iff $\forall\pi$ from $s_0: \ \pi \models \mathbf{X}g$
(P3*)	iff $\forall\pi: \ s_1 \models g$
(S1)	iff $\forall\pi: g \in L(s_1)$ iff $g \in AP$

Figure 4: Example of CTL formula.

computation path.

In **Computation Tree Logic** (CTL) the temporal operators—either **X** or **U**— must be directly preceded by a path quantifier—either **A** or **E**.

Syntax
The logic CTL is obtained by restricting the syntax to disallow boolean combinations of path formulas and nesting of temporal operators. Formally, we replace the CTL* rules (P1–3) by

(P3*) If f, g are the state formulas then $\mathbf{X}f$ and $f\mathbf{U}g$ are path formulas.

The set of state formulas generated by rules (S1–3) and (P3*) forms the CTL. The other CTL formulas can be introduced as abbreviations given by observation 1.

Semantics
A state formulas of CTL is interpreted using the rules (S1–3) of the CTL* semantics. The rule (P3*) is used for path formulas:

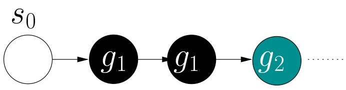
$$\text{(P3*)} \quad \pi \models \mathbf{X}f_1 \quad \Leftrightarrow \quad s_1 \models f_1$$
$$\pi \models f_1\mathbf{U}f_2 \quad \Leftrightarrow \quad \exists k \geq 0 : s_k \models f_2 \text{ and } \forall 0 \leq j < k : s_j \models f_1$$

In figure 4 we show how to apply these semantic rules to check whether CTL formula $\mathbf{AX}g$ is true for some Kripke structure M.

Below, we give some CTL formulas that typically arise in the verification of concurrent systems.

- **Invariance: AG** *property* — for all computations globally (i.e. in each state) some *property* holds.

- **Reachability: EF** *property* — it is possible eventually to get to a state with the given *property*.



$M, \pi \models \mathbf{X}(g_1 \mathbf{U} g_2)$ true iff:

(P3) $\pi \models \mathbf{X}(g_1 \mathbf{U} g_2)$ iff $\pi^1 \models g_1 \mathbf{U} g_2$
(P3) iff $\exists k \geq 1 : \pi^k \models g_2$ and $\forall 1 \leq j < k : \pi^j \models g_1$
(P1*) iff $\exists k \geq 1 : g_2 \in L(s_k)$ and $\forall 1 \leq j < k : g_1 \in L(s_j)$

Figure 5: Example of LTL formula.

- **Recovery: AG EF** *restart* — from any state it is possible to get to the *restart* state.
- **Inevitability: AF***stable* — for all computation eventually, the system reaches a *stable* state.
- **Non starvation:** $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$[1] — if a request occurs, then it will be eventually acknowledged.

Linear Temporal Logic (LTL) consists of formulas which allow to describe properties along a single path.

Syntax

The set of path formula generated by the following rules (P1*), (P2) and (P3) forms the language of LTL.

(P1*) Each atomic proposition p is a path formula.

(P2) If f, g are path formulas then so are $\neg f$ and $f \vee g$.

(P3) If f, g are path formulas then so are $\mathbf{X}f$ and $f\mathbf{U}g$.

Semantics

For path formulas g_1, g_2 and the path $\pi =< s_0, s_1, \ldots >$ we define inductively the relation $\models$ as follows:

$$\begin{array}{llcl}
\text{(P1*)} & \pi \models g_1 & \Leftrightarrow & g_1 \in L(s_0) \text{ iff } g_1 \in AP \\
\text{(P2)} & \pi \models \neg g_1 & \Leftrightarrow & \text{if not } \pi \models g_1 \\
 & \pi \models g_1 \vee g_2 & \Leftrightarrow & \pi \models g_1 \text{ or } \pi \models g_2 \\
\text{(P3)} & \pi \models \mathbf{X} g_1 & \Leftrightarrow & \pi^1 \models g_1 \\
 & \pi \models g_1 \mathbf{U} g_2 & \Leftrightarrow & \exists k \geq 0 : \pi^k \models g_2 \text{ and } \forall 0 \leq j < k : \pi^j \models g_1
\end{array}$$

Figure 5 shows LTL formula $M, \pi \models \mathbf{X}(g_1 \mathbf{U} g_2)$ that is true along some path π starting from state s_0 in Kripke structure M.

[1] $a \rightarrow b$ denotes the implication which is defined in the usual way as $a \rightarrow b \equiv \neg a \vee b$

5 Expressiveness of CTL, LTL and CTL*

It was shown in [3] and [4] that the three logics that we have discussed have different expressive powers.

Definition 3 *We say that L_1 is strictly less expressive than L_2, i.e. $L_1 < L_2$, iff the following two properties hold:*

(a) Any formula p in the logic L_1 can be expressed by a formula q of logic L_2:
$\forall p \in L_1 : \exists q \in L_2 : \forall M, s : \; M, s \models p \textit{ iff } M, s \models q$

(b) The converse of (a) does not hold some formula:
$\exists q \in L_2 : \forall p \in L_1 : \exists M, s : \neg(M, s \models q \textit{ iff } \; M, s \models p)$

We have the following results about the expressive power of CTL*, LTL and CTL:

- LTL < CTL*
  The CTL* formula $\mathbf{EF}p$ is not equivalent to any LTL formula [3].

- CTL < CTL*
  The CTL* formula $\mathbf{A}[\mathbf{F}(p \rightarrow \mathbf{AX}p)]$ is not equivalent to any CTL formula [3].

- CTL and LTL are not comparable, i.e. CTL is not strictly less expressive than LTL and vice versa: $\neg$(CTL<LTL) $\wedge$ $\neg$(LTL<CTL). This was shown in [4].

The correlation between the expressiveness of the above logics is represented in figure 6.

6 Fairness

Finally, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along so-called fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we wish to consider only those computations in which the arbiter does not ignore one of its request inputs forever. Such properties can be expressed by an LTL or CTL* formula, for example $\mathbf{GF}f$, but cannot be directly expressed in CTL. In order to deal with fairness in CTL we modify its semantics slightly. We call the new semantics of the logic *fair semantics*.

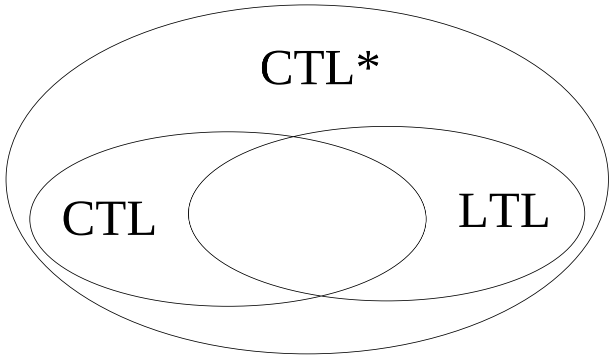


Figure 6: Expressiveness of temporal logics

Definition 4 *A fair Kripke structure is a 5-tuple* $M = (S, S_0, R, L, F)$ *where*

- S, S_0, R *and* L *are as in Definition 1.*
- $F \subseteq 2^S$ *is an additional set of fairness constraints, where a fairness constraint is an arbitrary set of states in Kripke structure.*

Let $\pi =< s_0, s_1, \ldots >$ be a path in M. With the $\inf(\pi)$ we denote the set of states which appears infinitely often on the path π, i.e.:

$\inf(\pi) = \{s \mid s = s_i \text{ for infinitely many } i\}$.

We say that a path π is *fair* iff for each fairness constraint $P \in F$ some state of P appears infinitely often on the path π, i.e.: $\inf(\pi) \cap P \neq \varnothing$. We will write $M, s \models_F f$ to indicate that the state formula f is true in the state s of the fair Kripke structure M. Similarly, we write $M, \pi \models_F g$ to indicate that the path formula g is true along a fair path π in M. Only the rules (S1), (S3) of the original semantics of CTL change:

(S1)	$s \models_F f_1$	$\Leftrightarrow$	$\exists$ a fair path π starting from s and $f_1 \in L(s)$
(S3)	$s \models_F \mathbf{E}g_1$	$\Leftrightarrow$	$\exists$ a fair path π starting from s such that $\pi \models_F g_1$
	$s \models_F \mathbf{A}g_1$	$\Leftrightarrow$	$\forall$ fair paths π starting from s, $\pi \models_F g_1$

References

[1] Clarke E., Grumberg O., Peled D.: Model Checking (1999)

[2] Emerson E.: Temporal and Modal Logic, in: van Leeuwen – Handbook of Theoretical Computer Science, Volume B (1990)

[3] Emerson E., Halpern J.: "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic, in: Journal of the ACM, Vol.33. No.I. January (1986)

[4] Lamport. L.: "Sometime" is sometimes "not never" - On the temporal logic of programs, in: Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages. ACM, New York (1980)