

## Verification Using Temporal Logic

### Sources:

- E.M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, Cambridge, 2000.
- E.A. Emerson. “Temporal and Modal Logic.” Chapter 16 in Volume B of J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. MIT Press, Cambridge, 1990.
- D. Peled. *Software Reliability Methods*. Springer-Verlag, Berlin, 2001.

## Temporal Logic ...

- ... a language for describing time-varying properties of systems.
- ... originally developed as a branch of philosophical logic.
- ... is a *modal* logic (logic of possibility and necessity).
- ... brought to attention of computer scientists by A. Pnueli in 1977.

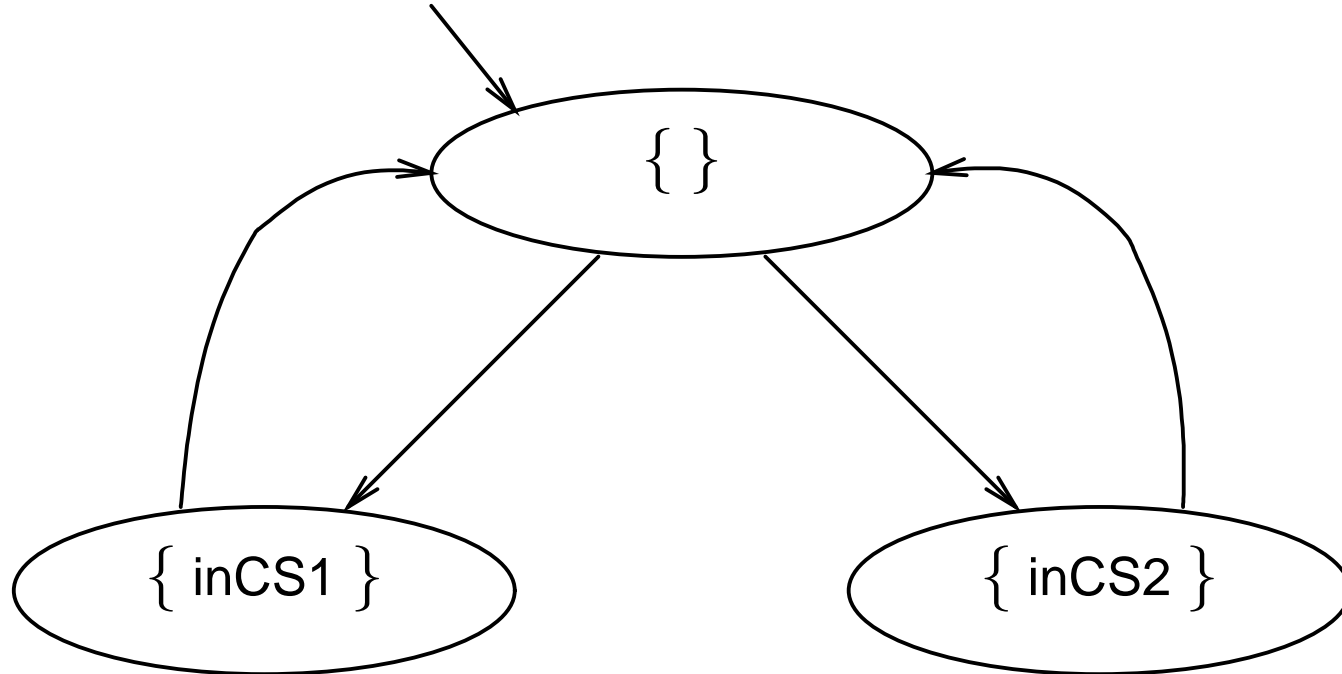
## Temporal Logic Verification Framework

How is the verification question  $S \text{ sat } R$  formulated in the temporal-logic setting?

- Temporal logic used to define requirements  $Spec$ .
- $Sys? \text{ sat?}$

## Kripke Structures

*Kripke structures used as system models in temporal logic.*

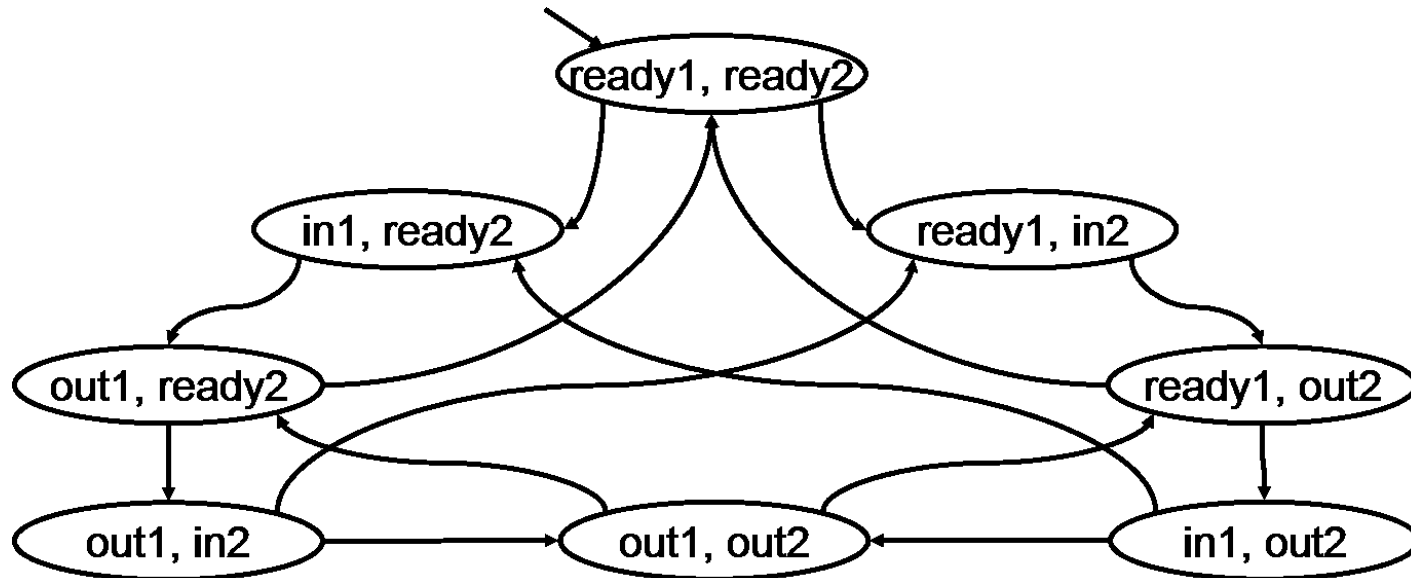


## Defining Kripke Structures

Mathematically, a Kripke structure has form  $\langle S, \mathcal{A}, R, \ell, s_I \rangle$ , where:

- $S$  is a set of *states*;
- $\mathcal{A}$  is a set of *atomic propositions*;
- $R \subseteq S \times S$  is the *transition relation*;
- $\ell : S \rightarrow 2^{\mathcal{A}}$  is the *labeling*; and
- $s_I \in S$  is the *initial state*.

### Another Kripke Structure Example



## Where Do Kripke Structures Come From?

Kripke structures are usually not specified directly by users; they are “compiled” from higher-level specifications using (small-step) *operational semantics* of programming / modeling notation.

- States: assignments of variables to values
- Labeling function given as relation on states / atomic propositions
- Transitions: execution steps defined by semantics

## Example: UNITY

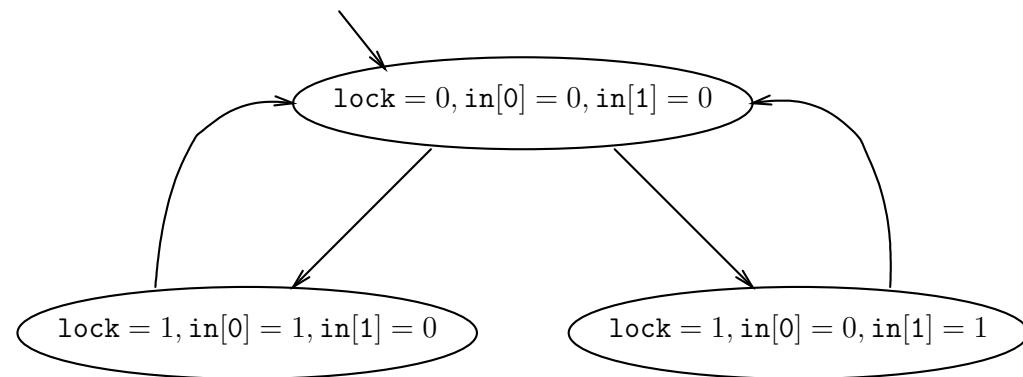
- A modeling notation for concurrent systems
- See K.M. Chandy and J. Misra, *Parallel Program Design*, Addison Wesley, Reading MA, 1988
- Emblematic of notations used by model-checking tools

## Example UNITY Model

```

program mutex
declare
  in: array [0..1] of int
  lock: int
initially
  in[0] = in[1] = lock = 0
assign
  P0enter: lock = 0 → lock, in[0] = 1, 1
  P1enter: lock = 0 → lock, in[1] = 1, 1
  P0exit: in[0] = 1 → lock, in[0] = 0, 0
  P1exit: in[1] = 1 → lock, in[0] = 0, 0
end

```



## UNITY in Detail

- Programs consist of `declare`, `initially`, `assign` sections (among others)
- `declare` section defines variables
- `initially` section initializes variables
- `assign` section contains guarded commands
  - Guarded command has form  $G \rightarrow S$
  - $G$  is boolean condition,  $S$  is Hoare program

Intended semantics: at each step, command with true guard is selected, executed atomically.

Program terminates when all guards are false.

## Defining Temporal Logic

Now that we know what systems (*Sys*) are, what about requirements (*Spec*)?

**Idea** Formulas of temporal logic constitute *Spec*.

⇒ but there are a number of variants of temporal logic.

An important categorization: *linear time* vs. *branching time*. The difference: how time is modeled!  
(More on this later.)

We'll start with linear-time.

## Linear-Time Temporal Logic (LTL)

Used to specify properties of *sequences*.

Given: set  $(a, b \in) \mathcal{A}$  of atomic propositions. Formulas generated as follows.

$$\begin{array}{l} \phi ::= a \\ | \neg\phi \\ | \phi \vee \phi \\ | X\phi \quad \text{"next"} \\ | \phi \text{ U } \phi \quad \text{"until"} \end{array}$$

$\Phi_{\text{LTL}}$  is the set of all LTL formulas.

## LTL Semantics

Formal semantics given as  $\models \subseteq (2^{\mathcal{A}})^{\omega} \times \Phi_{\text{LTL}}$ .

- $\mathcal{A}$ : set of atomic propositions
- $2^{\mathcal{A}}$ : set of all subsets of  $\mathcal{A}$ .
- $(2^{\mathcal{A}})^{\omega}$ : set of infinite sequences of  $2^{\mathcal{A}}$ .

That is, formulas interpreted with respect to sequences of sets of atomic propositions.

$\pi \models \phi$ : “sequence  $\pi$  has property  $\phi$ ”

## Notation

If  $\pi = A_0A_1 \dots \in (2^{\mathcal{A}})^\omega$  then

- $\pi[i] = A_i \in 2^{\mathcal{A}}$
- $\pi[i..] = A_iA_{i+1} \dots \in (2^{\mathcal{A}})^\omega$

## LTL Semantics (cont.)

$\pi \models a$  if  $a \in \pi[0]$ .

$\pi \models \neg\phi$  if  $\pi \not\models \phi$ .

$\pi \models \phi_1 \vee \phi_2$  if  $\pi \models \phi_1$  or  $\pi \models \phi_2$ .

$\pi \models X\phi$  if  $\pi[1..] \models \phi$ .

$\pi \models \phi_1 \text{ U } \phi_2$  if

$\exists i \geq 0. \pi[i..] \models \phi_2$  and

$\forall j < i. \pi[j..] \models \phi_1$

## Derived Operators

$$\mathbf{tt} \triangleq a \vee \neg a \quad \textit{true}$$

$$\mathbf{ff} \triangleq \neg \mathbf{tt} \quad \textit{false}$$

$$\phi_1 \wedge \phi_2 \triangleq \neg((\neg\phi_1) \vee (\neg\phi_2)) \quad \textit{conjunction}$$

$$\phi_1 \Rightarrow \phi_2 \triangleq (\neg\phi_1) \vee \phi_2 \quad \textit{implication}$$

## More Derived Operators

*Eventually*  $F\phi$

$t \text{ U } \phi$

*Always*  $G\phi$

$\neg F\neg\phi$

## Still More Derived Operators

*Release*      $\phi_1 \text{ R } \phi_2$   
                  $\neg((\neg\phi_1) \text{ U } (\neg\phi_2))$

*Weak Until*      $\phi_1 \text{ W } \phi_2$   
                  $\phi_2 \text{ R } (\phi_1 \vee \phi_2)$

What does  $\neg((\neg\phi_2) \text{ U } (\neg\phi_1 \wedge \neg\phi_2))$  mean?

## Writing Requirements in LTL

LTL can be used to specify different properties of “execution sequences”.

**Safety** “Nothing bad ever happens.”

$G \neg bad$

**Liveness** “Something good eventually happens.”

$F good$

## More Kinds of Properties

**Responsiveness** “Every request is eventually serviced.”

$G (req \Rightarrow F \text{ serviced})$

**Reactivity**

$(GF \text{ enabled}) \Rightarrow (GF \text{ executed})$

**Partition**

$\phi_1 \cup (\phi_2 \cup \phi_3)$

## Example: Specifying Mutual Exclusion

$\mathcal{A} = \{in_i, out_i, trying_i\}, i = 1, 2.$

Mutual Exclusion

$G \neg(in_1 \wedge in_2)$

Starvation-Freedom for Process 1

$G(trying_1 \Rightarrow F in_1)$

1-Bounded Overtaking for Process 1

$G(trying_1 \Rightarrow (trying_2 \cup in_2 \cup in_1))$

## But What About System Properties?

So far we have used LTL to define properties of “executions”.

How do we use it to define properties/requirements of systems?

**Idea** Define a system to satisfy a formula if “all its executions” do.

How to make this precise?

## Formalizing sat

Let  $M = \langle S, \mathcal{A}, R, \ell, s_I \rangle$  be a Kripke structure.

- $\pi = s_0 s_1 \dots \in S^\omega$  is an *execution* of  $M$  if:
  - $s_0 = s_I$
  - For all  $i \geq 0$ ,  $\langle s_i, s_{i+1} \rangle \in R$  or for all  $s \in S$ ,  $\langle s_i, s \rangle \notin R$  and  $s_{i+1} = s_i$ .

$E(M)$ : set of all executions of  $M$ .

- $\ell(s_0 s_1 \dots) = \ell(s_0) \ell(s_1) \dots$
- $M \text{ sat } \phi$  if for all  $\pi \in E(M)$ ,  $\ell(\pi) \models \phi$ .

## What We Have...

... a verification framework based on LTL!

Systems:            Kripke structures

Requirements:    LTL formulas

sat:                sat

What's next?

1. Other temporal logics.
2. How to show  $M \text{ sat } \phi$ .

## Branching-Time Temporal Logic

Recall:

- Two kinds of temporal logic: linear-time and branching-time
- Linear-time: models are sequences
- Branching-time: models are “trees” (alternatively, states in Kripke structures)

How do we define a branching-time temporal logic?

**CTL\* = LTL + Path Quantifiers**

The CTL\* approach: add “path quantifiers” to LTL!

$E\phi$ : satisfied by a state if there exists a path from the state and satisfying  $\phi$ .

$E$  is a *path quantifier*.

## Syntax of CTL\*

Defines *state* formulas ...

$$\begin{array}{l} \sigma ::= a \\ \quad | \neg\sigma \\ \quad | \sigma \vee \sigma \\ \quad | E\phi \end{array}$$

$\Sigma_{\text{CTL}^*}$ : set of all CTL\* (state) formulas.

## Syntax of CTL\* (cont.)

... and *path* formulas.

$$\begin{array}{l} \phi ::= \sigma \\ | \neg \phi \\ | \phi \vee \phi \\ | X \phi \\ | \phi U \phi \end{array}$$

$\Phi_{\text{CTL}^*}$ : set of all CTL\* path formulas.

## Defining Semantics of CTL\*

... given with respect to Kripke structure  $M = \langle S, \mathcal{A}, R, \ell, s_I \rangle$  as relation

$$\models_M \subseteq (S \times \Sigma_{\text{CTL}^*}) \cup (S^\omega \times \Phi_{\text{CTL}^*})$$

... i.e. states related to state formulas and paths to path formulas.

Fix  $M$  in what follows.

**Notation**  $E(M, s) \subseteq S^\omega$ : execution paths emanating from  $s$ .  $\pi \in E(M, s)$  if:

- $\pi[0] = s$
- For all  $i \geq 0$ ,  $\langle \pi[i], \pi[i+1] \rangle \in R$  or for all  $s' \in S$ ,  $\langle \pi[i], s' \rangle \notin R$  and  $\pi[i+1] = \pi[i]$ .

## Semantics of State Formulas

$s \models_M a$  if  $a \in \ell(s)$ .

$s \models_M \neg\sigma$  if  $s \not\models_M \sigma$ .

$s \models_M \sigma_1 \vee \sigma_2$  if  $s \models_M \sigma_1$  or  $s \models_M \sigma_2$ .

$s \models_M E\phi$  if there exists  $\pi \in E(M, s)$  such that  $\pi \models_M \phi$ .

## Semantics of Path Formulas

$\pi \models_M \sigma$  if  $\pi[0] \models_M \sigma$ .

$\pi \models_M \neg\phi$  if  $\pi \not\models_M \phi$ .

$\pi \models_M \phi_1 \vee \phi_2$  if  $\pi \models_M \phi_1$  or  $\pi \models_M \phi_2$ .

$\pi \models_M X\phi$  if  $\pi[1..] \models_M \phi$ .

$\pi \models_M \phi_1 \text{ U } \phi_2$  if  $\exists i \geq 0. \sigma[i..] \models_M \phi_2$  and  $\forall j < i. \sigma[j..] \models_M \phi_1$

Note:  $\neg, \vee, X, \text{U}$  as in LTL!

Derived operators: usual LTL derived operators,  $A \equiv \neg E \neg$ .

## CTL\* Verification Framework

**Sys:** Kripke structures  $M = \langle S, \mathcal{A}, R, \ell, s_I \rangle$

**Spec:** CTL\* *state* formulas  $\sigma \in \Sigma_{\text{CTL}^*}$

**sat:**  $M$  sat  $\sigma$  if  $s_I \models_M \sigma$ .

## Expressing Properties in CTL\*

**Possibility** “Any message can be lost.”

$AG (sent \Rightarrow EF lost)$

**LTL** Any LTL system specification  $\phi$  can be rendered as a CTL\* state formula  $A\phi$ .

So CTL\* is at least as expressive a system specification notation as LTL. In fact, it is more so (LTL can't express E).

## CTL ...

... a sublogic of CTL\*

... development preceded that of CTL\*

... stands for “Computation Tree Logic”

... first temporal logic used in model checking

CTL formulas: state formulas in CTL\* in which every path modality (U, X, G, etc.) is immediately preceded by a path quantifier.

CTL  $AG(sent \Rightarrow AF\ received)$

Not CTL  $AG(sent \Rightarrow F\ received)$

## Formal CTL Syntax

**Note** All CTL formulas are *state* formulas.

$$\begin{array}{l} \sigma ::= a \\ \quad | \neg\sigma \\ \quad | \sigma \vee \sigma \\ \quad | EX\sigma \\ \quad | E(\sigma U \sigma) \\ \quad | E(\sigma R \sigma) \end{array}$$

$\Sigma_{\text{CTL}}$ : set of all CTL formulas. Recall that R is dual of U.

Derived operators: AX, AU, AR, EG, EF, AG, AF, etc.

## Expressiveness of CTL System Specifications

- No more expressive than  $CTL^*$ , since  $\Sigma_{CTL} \subseteq \Sigma_{CTL^*}$ .
- Incomparable to LTL:
  - CTL can express possibility properties.
  - LTL can express “fairness” properties.  
AFG  $a$  expressible in LTL, not in CTL.
- Consequently, strictly less expressive than  $CTL^*$ ! (Why?)