

# Part VI

## Model Checking

# 1. Introduction to Model-Checking

- **Idea:**
  - system **specified** by (temporal) logic **formula**
  - **implementation** of the system represented by **transition system**
  - **check**, whether implementation is **model** of the formula,  
i.e. all states fulfill the formula
- **Application:**
  - today usual for: verification of hardware, communication protocols,  
device drivers . . .
  - in the future (?): application systems
  - in particular: distributed systems

## Properties of Model-Checking

- **Advantage:**
  - **proves** program properties completely (rather than for examples)
  - **fully automatic** ( $\rightarrow$  usable by everyone)
  - delivers **counter example**, if the specification is not met
- **Disadvantage:**
  - **explosion of the state space** (and hence of the computation costs) for larger systems

## 2. Modelling Transition Systems

- Focus: reactive systems
  - (in general) distributed, non-terminating
  - often non-deterministic
  - cannot be adequately described just by I/O behaviour
  - e.g. operating system, e-shop, e-banking, ...
  - computation  $\hat{=}$  infinite sequence of states in transition system

## Kripke Structure

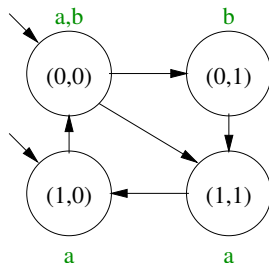
### Definition (6.1)

- let  $AF$  be a set of atomic formulae
- a **Kripke structure** is a 4-tuple  $K = (S, S_0, \rightarrow, \lambda)$  with:
  - $S$ : finite set of **states**
  - $S_0 \subset S$ : set of **initial states**
  - $\rightarrow \subset S \times S$ : total **state transition relation**,  
i.e.  $\forall s \in S \exists s' \in S s \rightarrow s'$
  - $\lambda : S \rightarrow \mathcal{P}(AF)$  **labeling function** ( $\mathcal{P}$ : power set)
- a **path** in  $K$  starting from  $s_0$  is an infinite sequence  $\pi = s_0 s_1 s_2 \dots$   
of states with  $s_i \rightarrow s_{i+1}$  for  $i \in \mathbb{N}$  □

## Example: Kripke-Structure

- $K = (\{(0,0), (0,1), (1,0), (1,1)\}, \{(0,0), (1,0)\}, \rightarrow, \lambda)$
- $\rightarrow =$   
 $\{((0,0), (0,1)), ((0,0), (1,1)), ((0,1), (1,1)), ((1,1), (1,0)), ((1,0), (0,0))\}$

$s$	$\lambda(s)$
(1, 1)	{a}
(0, 1)	{b}
(1, 0)	{a}
(0, 0)	{a, b}



## Logic Representation of a Transition System

- a Kripke structure can be described by a logic formula
- later: show that this formula implies the specification
- let  $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$  be a Kripke structure
- let  $X$  be the set of variables in the system
- each state  $s \in \mathcal{S}$  is characterized by a corresponding variable valuation  $\beta_s : X \rightarrow D$
- the formula  $F_s := \bigwedge_{i \in \mathbb{N}} x_i = \beta_s(x_i)$  characterizes  $s \in \mathcal{S}$
- a formula  $F$  can be understood as description of the set  $\mathcal{S}_F$  of states that fulfill it:  $\mathcal{S}_F = \{s \in \mathcal{S} \mid \beta_s \models F\}$

## Logic Representation of a Transition System (2)

- let  $X'$  be a copy of  $X$  (“variables in subsequent state”)
- let  $\beta'_s : X' \rightarrow D$  be a corresponding variable valuation with  $\beta'_s(x') = \beta_s(x)$  for  $x \in X$  and  $x' \in X'$
- the **transition relation**  $\rightarrow$  can be described by a formula  $F_{\rightarrow}$  over  $X$  and  $X'$  for which holds:

$$\forall s_1, s_2 \in S \quad s_1 \rightarrow s_2 \quad \Rightarrow \quad \beta_{s_1} \cup \beta'_{s_2} \models F_{\rightarrow}$$

- let  $AF$  be a set of atomic propositions, typically of the form  $x = d$  with  $x \in X$  and  $d \in D$
- trivially:  $\beta_s \models x = d$ , if  $\beta_s(x) = d$  for  $s \in S$

## Construction of a Kripke Structure

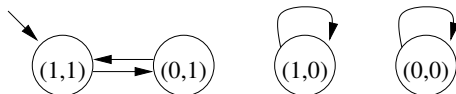
- let  $X$  be the set of **system variables**
- the formula  $F_0$  describes the set of **initial states**
- the formula  $F_T$  describes the **state transition relation**
- $F_0$  and  $F_T$  determine the **Kripke structure**  $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$ :
  - $\mathcal{S} = \{\beta \mid \beta : X \rightarrow D\}$
  - $\mathcal{S}_0 = \mathcal{S}_{F_0}$
  - $s \rightarrow s'$ , if  $\beta_s \cup \beta'_{s'} \models F_T$  for  $s, s' \in \mathcal{S}$
  - $s \rightarrow s$ , if  $\nexists s' \in \mathcal{S} \quad \beta_s \cup \beta'_{s'} \models F_T$  ( $\rightarrow$  total!)
  - $\lambda(s) = \{a \in AF \mid \beta_s \models a\}$

## Example: Construction of a Kripke Structure

- initial program: `while true do x := (x+y) mod 2`
- $F_0 = x = 1 \wedge y = 1$
- $F_T = x' = (x + y) \bmod 2 \wedge y' = y$
- $K = (\{0, 1\}^2, \{(1, 1)\}, \rightarrow, \lambda)$

with  $\rightarrow = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\}$

$s$	$\lambda(s)$
(1, 1)	$\{x = 1, y = 1\}$
(0, 1)	$\{x = 0, y = 1\}$
(1, 0)	$\{x = 1, y = 0\}$
(0, 0)	$\{x = 0, y = 0\}$



## Granularity of Transitions

- the granularity of the modelling has to be chosen appropriately
- otherwise errors can be overlooked or reported erroneously

### Example:

coarse granularity:

$$\alpha: \quad x := x + y \quad ||$$

$$\beta: \quad y := y + x$$

fine granularity:

$$\alpha_0: \quad \text{load } R_1, x \quad \beta_0: \quad \text{load } R_2, y$$

$$\alpha_1: \quad \text{add } R_1, y \quad || \quad \beta_1: \quad \text{add } R_2, x$$

$$\alpha_2: \quad \text{store } R_1, x \quad \beta_2: \quad \text{store } R_2, y$$

- starting from an initial state with  $x = 3 \wedge y = 2$  states we reach
  - with coarse granularity:  $x = 5 \wedge y = 7$  or  $x = 8 \wedge y = 5$
  - with fine granularity: additionally  $x = 5 \wedge y = 5$   
(overlooked or erroneously reported?)

## Synchronous and Asynchronous Processes

- a distributed system can work synchronously or asynchronously
- **synchronous**: all processes execute one step simultaneously (e.g. switching circuits)
- **asynchronous**: one (non-deterministically) selected process executes a step (interleaving; typical for software)
- let a system  $S = P_1 || \dots || P_n$  consist of  $n$  concurrent processes
- let the subformula  $F_i$  describe the state transition in  $P_i$
- then the **state transition relation** in  $S$  can be described by:
  - $F_T = \bigwedge_{i=1}^n F_i$ , if synchronous
  - $F_T = \bigvee_{i=1}^n F_i$ , if asynchronous

## Transforming a Program into a Logic Formula

- extension of IMP: a program  $P$  can:
  - consist of several concurrent processes, i.e.

$P = \text{cobegin } c_1 \parallel \dots \parallel c_n \text{ coend}$

- contain the synchronization statements  $\text{wait}(b)$ ,  $\text{lock}(x)$  and  $\text{unlock}(x)$
- the transformation of an asynchronous program works in 2 steps:
  - step 1: transformation  $\mathcal{L}$  of a program into a program with labels
  - step 2: transformation  $\mathcal{T}$  of a program with labels into a formula

## 1) Transformation into a Program with Labels

- $\mathcal{L}(x := e) = x := e$  (analogously for other atomic statements such as `skip`, ...)
- $\mathcal{L}(c_1, c_2) = \mathcal{L}(c_1); l_1 : \mathcal{L}(c_2)$
- $\mathcal{L}(\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}) =$   
     $\text{if } b \text{ then } l_1 : \mathcal{L}(c_1) \text{ else } l_2 : \mathcal{L}(c_2) \text{ fi}$
- $\mathcal{L}(\text{while } b \text{ do } c \text{ od}) = \text{while } b \text{ do } l_1 : \mathcal{L}(c) \text{ od}$
- here  $l_1, l_2$  are “new” labels
- moreover a new label is added at the beginning and end of each program / process

## Example: Transformation into a Program with Labels

- let  $P_{\text{turn}} = \text{cobegin } c_1 \parallel c_2 \text{ coend}$
- with  $c_1 = \text{while true do wait (turn == 0); turn := 1 od}$
- and  $c_2 = \text{while true do wait (turn == 1); turn := 0 od}$
- then:  $\mathcal{L}(P_{\text{turn}}) = l_0 : \text{cobegin } \mathcal{L}(c_1) \parallel \mathcal{L}(c_2) \text{ coend } l_3$  with:
  - $\mathcal{L}(c_1) = l_{10} : \text{while true do } l_{11} : \text{wait (turn == 0); } l_{12} : \text{turn := 1 od } l_{13}$
  - $\mathcal{L}(c_2) = l_{20} : \text{while true do } l_{21} : \text{wait (turn == 1); } l_{22} : \text{turn := 0 od } l_{23}$

## 2a) Transforming a Statement into a Formula (1)

- let  $l : c$  be a statement with labels and set of variables  $X$
- let  $pc \in X$  be a special variable over  $L \cup \{?\}$  (program counter),

where  $L$  is a set of labels

- abbreviation:  $same(Y) := \bigwedge_{x \in Y} x' = x$  for  $Y \subset X$

## Transforming a Statement into a Formula (2)

- for a labelled statement  $l : c$  and a successor label  $l'$

$\mathcal{I}_{pc}(l : c, l')$  is defined as follows:

$$\mathcal{I}_{pc}(l : x := e, l') = pc = l \wedge pc' = l' \wedge x' = e \wedge \text{same}(X - \{x, pc\})$$

$$\mathcal{I}_{pc}(l : \text{skip}, l') = pc = l \wedge pc' = l' \wedge \text{same}(X - \{pc\})$$

(analogously: `wait, ...`)

$$\mathcal{I}_{pc}(l : c_1; l'' : c_2, l') = \mathcal{I}_{pc}(l : c_1, l'') \vee \mathcal{I}_{pc}(l'' : c_2, l')$$

$$\begin{aligned} \mathcal{I}_{pc}(l : \text{if } b \text{ then } l_1 : c_1 \text{ else } l_2 : c_2 \text{ fi}, l') = \\ (pc = l \wedge pc' = l_1 \wedge b \wedge \text{same}(X - \{pc\})) \vee \\ (pc = l \wedge pc' = l_2 \wedge \neg b \wedge \text{same}(X - \{pc\})) \vee \\ \mathcal{I}_{pc}(l_1 : c_1, l') \vee \mathcal{I}_{pc}(l_2 : c_2, l') \end{aligned}$$

## Transforming a Statement into a Formula (3)

$\mathcal{T}_{pc}(l : \text{while } b \text{ do } l_1 : c_1 \text{ od}, l') =$

$(pc = l \wedge pc' = l_1 \wedge b \wedge \text{same}(X - \{pc\})) \vee$

$(pc = l \wedge pc' = l' \wedge \neg b \wedge \text{same}(X - \{pc\})) \vee$

$\mathcal{T}_{pc}(l_1 : c_1, l)$

## 2b) Transforming a Program into a Formula

- let  $P = l_0 : \text{cobegin } c_1 \parallel \dots \parallel c_n \text{ coend } l_{n+1}$  be a lbl. program
- let  $PC = \{pc, pc_1, \dots, pc_n\} \subset X$  be the set of program counters
- let  $X_j \subset X$  be the set of variables in  $c_j$  (including  $pc_j$ )
- let  $F_0$  have the form

$$F_0 = \bigwedge_{x \in X - PC} x = d_x \wedge pc = l_0 \wedge \bigwedge_{i=1}^n pc_i = ? \quad \text{with } d_x \in D$$

$$F_T := \mathcal{T}(l_0 : \text{cobegin } l_1 : c_1 l'_1 \parallel \dots \parallel l_n : c_n l'_n \text{ coend } l_{n+1}, l') =$$

$$(pc = l_0 \wedge \bigwedge_{i=1}^n pc'_i = l_i \wedge pc' = ? \wedge \text{same}(X - PC)) \vee$$

$$(pc = ? \wedge \bigwedge_{i=1}^n pc_i = l'_i \wedge pc' = l' \wedge \bigwedge_{i=1}^n pc'_i = ? \wedge \text{same}(X - PC)) \vee$$

$$\left( \bigvee_{i=1}^n (\mathcal{T}_{pc_i}(l_i : c_i, l'_i)) \wedge \text{same}(X - X_i) \right)$$

## Treatment of Synchronization Statements

$$\mathcal{T}_{pc_i}(I: \text{wait}(b), I') =$$

$$(pc_i = I \wedge pc'_i = I \wedge \neg b \wedge \text{same}(X_i - \{pc_i\})) \vee$$

$$(pc_i = I \wedge pc'_i = I' \wedge b \wedge \text{same}(X_i - \{pc_i\}))$$

$$\mathcal{T}_{pc_i}(I: \text{lock}(x), I') =$$

$$(pc_i = I \wedge pc'_i = I \wedge x = 1 \wedge \text{same}(X_i - \{pc_i\})) \vee$$

$$(pc_i = I \wedge pc'_i = I' \wedge x = 0 \wedge x' = 1 \wedge \text{same}(X_i - \{x, pc_i\}))$$

$$\mathcal{T}_{pc_i}(I: \text{unlock}(x), I') = pc_i = I \wedge pc'_i = I' \wedge x' = 0 \wedge$$

$$\text{same}(X_i - \{x, pc_i\})$$

## Example: Transforming a Program into a Formula

- Initially:  $\mathcal{L}(P_{\text{turn}})$  (see above)

$$F_0 = pc = l_0 \wedge pc_1 = ? \wedge pc_2 = ?$$

$$F_T = (pc = l_0 \wedge pc'_1 = l_{10} \wedge pc'_2 = l_{20} \wedge pc' = ? \wedge \text{turn}' = \text{turn}) \vee$$

$$(pc = ? \wedge pc_1 = l_{13} \wedge pc_2 = l_{23} \wedge pc' = l_3 \wedge pc'_1 = ? \wedge pc'_2 = ?$$

$$\wedge \text{turn}' = \text{turn}) \vee$$

$$(\mathcal{I}_{pc_1}(l_{10} : c_1, l_{13}) \wedge pc' = pc \wedge pc'_2 = pc_2) \vee$$

$$(\mathcal{I}_{pc_2}(l_{20} : c_2, l_{23}) \wedge pc' = pc \wedge pc'_1 = pc_1)$$

## Example: Transforming a Program into a Formula (2)

$$\mathcal{T}_{pc_i}(l_{i0} : c_i, l_{i3}) = \quad (i = 1, 2)$$

$$(pc_i = l_{i0} \wedge pc'_i = l_{i1} \wedge \text{true} \wedge \text{turn}' = \text{turn}) \vee$$

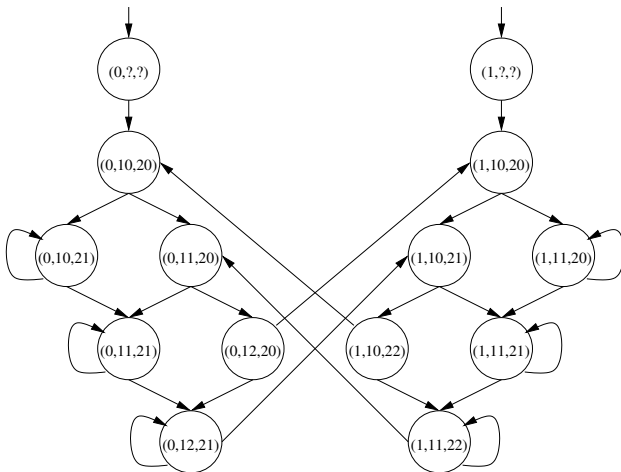
$$(pc_i = l_{i1} \wedge pc'_i = l_{i2} \wedge \text{turn} = i - 1 \wedge \text{turn}' = \text{turn}) \vee$$

$$(pc_i = l_{i2} \wedge pc'_i = l_{i0} \wedge \text{turn}' = i \bmod 2) \vee$$

$$(pc_i = l_{i1} \wedge pc'_i = l_{i1} \wedge \text{turn} \neq i - 1 \wedge \text{turn}' = \text{turn}) \vee$$

$$(pc_i = l_{i0} \wedge pc'_i = l_{i3} \wedge \text{false} \wedge \text{turn}' = \text{turn})$$

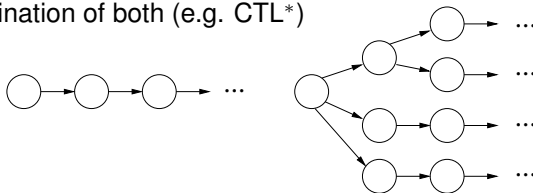
## Resulting Kripke Structure in the Example



- $(\text{turn}, pc_1, pc_2)$  ( $pc$  omitted, since everywhere  $pc=?$  except for initial states)
- only reachable states shown; deadlock free; starvation possible!

### 3. Temporal Logic

- for reactive systems in particular the timing behaviour is of interest
- relevant properties: liveness, safety, ...
- temporal logics are used to describe such properties of sequences of states
- we distinguish temporal logics with:
  - **linear time** (e.g. LTL)
  - **branching time** (e.g. CTL)
  - combination of both (e.g. CTL\*)



## Expressivity and Model-Checking Complexity

- there are properties which can be expressed in CTL but not in LTL and vice versa
- CTL\* subsumes CTL and LTL
- $\exists$  properties, which can be expressed in CTL\*, but neither in CTL nor in LTL
- a CTL formula  $F$  can be checked in linear time (i.e. in  $O(|F| + |\rightarrow|)$ )
- checking a LTL- or CTL\*-formula is PSPACE-complete (and hence at least NP-complete)

## Syntax of CTL\*

### Definition (6.2)

- syntax of a CTL\* (state-)formula  $f$  in EBNF:

$$f ::= a \mid \neg f \mid f \vee f \mid f \wedge f \mid E \varphi \mid A \varphi$$

- where  $a \in AF$  atomic formula and  $\varphi$  a CTL\* path formula with:

$$\varphi ::= f \mid \neg \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid X \varphi \mid F \varphi \mid G \varphi \mid \varphi U \varphi \mid \varphi R \varphi$$

□

## Names and Meanings of the CTL\* Operators

- **A**: (all) on all paths starting from the considered state
- **E**: (exists) for (at least) one path starting from the considered state
- **X**: (next) in the next state of the considered path
- **F**: (future, eventually) in (at least) one state of the considered path
- **G**: (globally, always) in all states of the considered path
- **U**: (until)  $\varphi_1 U \varphi_2$  holds, if  $\varphi_1$  holds on the considered path, until  $\varphi_2$  holds (at least) once.  $\varphi_2$  has to hold eventually.
- **R**: (release)  $\varphi_1 R \varphi_2$  holds, if  $\varphi_2$  holds on the considered path, until  $\varphi_1$  holds (at least) once. If  $\varphi_2$  holds forever,  $\varphi_1$  needs not hold at all.

## Semantics of CTL\*

### Definition (6.3)

- let  $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$  be a Kripke structure
- for  $\pi = s_0 s_1 \dots$ ,  $\pi^i$  denotes the suffix  $s_i s_{i+1} \dots$

$$s \models a \quad :\Leftrightarrow \quad a \in \lambda(s) \quad \text{for } a \in AF$$

$$s \models \neg f \quad :\Leftrightarrow \quad s \not\models f$$

$$s \models f_1 \vee f_2 \quad :\Leftrightarrow \quad s \models f_1 \text{ or } s \models f_2$$

$$s \models f_1 \wedge f_2 \quad :\Leftrightarrow \quad s \models f_1 \text{ and } s \models f_2$$

$$s \models E \varphi \quad :\Leftrightarrow \quad \exists \pi \text{ starting from } s \text{ with } \pi \models \varphi$$

$$s \models A \varphi \quad :\Leftrightarrow \quad \forall \pi \text{ starting from } s \quad \pi \models \varphi \text{ holds}$$

## Semantics of CTL\* (continued)

### Definition (6.3 (continued))

$$\pi \models f \quad :\Leftrightarrow \quad \pi = \mathbf{s} \dots \text{ and } \mathbf{s} \models f$$

$$\pi \models \neg\varphi \quad :\Leftrightarrow \quad \pi \not\models \varphi$$

$$\pi \models \varphi_1 \vee \varphi_2 \quad :\Leftrightarrow \quad \pi \models \varphi_1 \text{ or } \pi \models \varphi_2$$

$$\pi \models \varphi_1 \wedge \varphi_2 \quad :\Leftrightarrow \quad \pi \models \varphi_1 \text{ and } \pi \models \varphi_2$$

$$\pi \models \mathbf{X} \varphi \quad :\Leftrightarrow \quad \pi^1 \models \varphi$$

$$\pi \models \mathbf{F} \varphi \quad :\Leftrightarrow \quad \exists k \geq 0 \text{ with } \pi^k \models \varphi$$

$$\pi \models \mathbf{G} \varphi \quad :\Leftrightarrow \quad \forall k \geq 0 \text{ with } \pi^k \models \varphi$$

$$\pi \models \varphi_1 \mathbf{U} \varphi_2 \quad :\Leftrightarrow \quad \exists k \geq 0 \text{ with } \pi^k \models \varphi_2 \text{ and } \forall 0 \leq j < k \quad \pi^j \models \varphi_1$$

$$\pi \models \varphi_1 \mathbf{R} \varphi_2 \quad :\Leftrightarrow \quad \forall k \geq 0: \text{ if } \forall j < k \quad \pi^j \not\models \varphi_1, \text{ then } \pi^k \models \varphi_2 \quad \square$$

## Replacing Operators

- a CTL\* formula can be formulated by only using  $\neg, \vee, X, U$  and  $E$
- $f_1 \wedge f_2 = \neg(\neg f_1 \vee \neg f_2)$  (analogously for:  $\varphi_1 \wedge \varphi_2$ )
- $\varphi_1 R \varphi_2 = \neg(\neg\varphi_1 U \neg\varphi_2)$
- $F \varphi = \text{true } U \varphi$
- $G \varphi = \neg F(\neg\varphi)$
- $A \varphi = \neg E(\neg\varphi)$

## CTL

- CTL is a sublogic of CTL\* with branching time
- the temporal operators  $X$ ,  $F$ ,  $G$ ,  $U$  and  $R$  must be preceded by a path operator ( $A$  or  $E$ )
- **Advantage**: a CTL-formula can be checked in linear time

### Definition (6.4, syntax of CTL)

- the syntax of a CTL state formula corresponds to CTL\*
- a **CTL path formula** has the form:

$$\varphi ::= Xf \mid Ff \mid Gf \mid fUf \mid fRf$$

□

# LTL

- LTL is a sublogic of CTL\* with linear time
- path quantors are missing
- checking a LTL formula is PSPACE-complete

## Definition (6.5, syntax of LTL)

- syntax of a LTL formula  $f$  in EBNF:

$$f ::= A \varphi$$

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi U \varphi \mid \varphi R \varphi$$

- where  $a \in AF$  is an atomic formula □

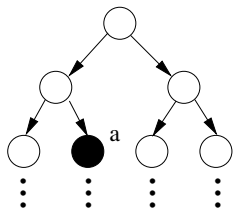
## Comparing the Expressivity of Temporal Logics

- the LTL formula  $A(FG a)$  cannot be expressed in CTL
- the CTL formula  $AG(EF a)$  cannot be expressed in LTL
- the CTL\*-formula  $A(FG a) \wedge AG(EF a)$  can neither be expressed in CTL nor in LTL
- in the sequel we will mainly focus on CTL

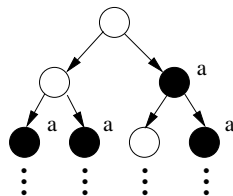
## Reducing the Number of Operators

- basic CTL operators:  $AX$ ,  $EX$ ,  $AF$ ,  $EF$ ,  $AG$ ,  $EG$ ,  $AU$ ,  $EU$ ,  $AR$ ,  $ER$
- they can be expressed in terms of  $EX$ ,  $EG$  and  $EU$ :
- $AX \varphi = \neg EX(\neg\varphi)$
- $EF \varphi = E(\text{true } U \varphi)$
- $AG \varphi = \neg EF(\neg\varphi)$
- $AF \varphi = \neg EG(\neg\varphi)$
- $A(\varphi_1 U \varphi_2) = \neg E(\neg\varphi_2 U (\neg\varphi_1 \wedge \neg\varphi_2)) \wedge \neg EG\neg\varphi_2$
- $A(\varphi_1 R \varphi_2) = \neg E(\neg\varphi_1 U \varphi_2)$
- $E(\varphi_1 R \varphi_2) = \neg A(\neg\varphi_1 U \varphi_2)$

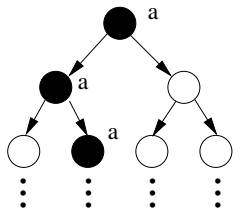
## Example: CTL-Formulae and Computation Trees



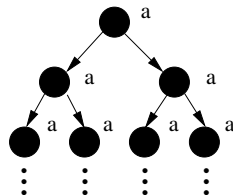
$$s_0 \models EF a$$



$$s_0 \models AF a$$



$$s_0 \models EG a$$



$$s_0 \models AG a$$

## Examples: CTL Formulae

- $EF(start \wedge \neg ready)$ :  
the system can start without being ready
- $AG(query \Rightarrow AF answer)$ :  
every query will eventually be answered
- $AG(AF deviceReady)$ :  
on every path, the device is ready infinitely often
- $AG(EF start)$ :  
it is always possible to return to the start

## 4. CTL Model Checking

- Basic Idea of Model Checking:
  - given: Kripke structure  $K = (\mathcal{S}, \mathcal{S}_0, \rightarrow, \lambda)$   
corresponding to a distributed system with finite state space
  - given: temporal-logic formula  $f$
  - compute the set  $\mathcal{S}_f$  of states fulfilling  $f$ , i.e.  $\mathcal{S}_f = \{s \in \mathcal{S} \mid s \models f\}$
  - check whether  $\mathcal{S}_0 \subseteq \mathcal{S}_f$

## CTL Model Checking: Basic Idea

- given: Kripke structure  $K = (S, S_0, \rightarrow, \lambda)$  and formula  $f$
- $f$  contains only the operators  $\neg, \vee, EX, EU$  and  $EG$  (after transformation)
- label each state  $s \in S$  with the subformulae of  $f$ , which  $s$  fulfills
- initially:  $\forall s \in S \text{ label}(s) := \lambda(s)$
- for  $i = 1, \dots, \text{nesting\_depth}(f)$ :
  - infer from the labels of iteration  $i - 1$ ,
  - which states fulfill which subformulae with nesting depth  $i$  and
  - label them correspondingly (\*)
- at the end:  $s \models f$  iff  $f \in \text{label}(s)$

## Case Distinction in Step (\*)

- case distinction based on the remaining operators  $(s \in S)$
- $f = \neg f_1$ :  $label(s) += \{f\}$ , if  $f_1 \notin label(s)$
- $f = f_1 \vee f_2$ :  $label(s) += \{f\}$ , if  $f_1 \in label(s)$  or  $f_2 \in label(s)$
- $f = EX f_1$ :  $label(s) += \{f\}$ , if  $\exists s' \in S$  with  $s \rightarrow s'$  and  $f_1 \in label(s')$
- $f = E(f_1 U f_2)$ : see procedure `checkEU`  $(O(|S| + |\rightarrow|))$
- $f = EG f_1$ : see procedure `checkEG`  $(O(|S| + |\rightarrow|))$

## Checking $E(f_1 U f_2)$

- starting from the states fulfilling  $f_2$ ,  
label preceding states fulfilling  $f_1$

procedure checkEU( $f_1, f_2$ )

$T := \{s \mid f_2 \in \text{label}(s)\};$

for all  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{E(f_1 U f_2)\};$

while  $T \neq \emptyset$  do

  choose  $s \in T$ ;

$T := T - \{s\};$

  for all  $s'$  such that  $s' \rightarrow s$  do

    if  $E(f_1 U f_2) \notin \text{label}(s')$  and  $f_1 \in \text{label}(s')$  then

$\text{label}(s') := \text{label}(s') \cup \{E(f_1 U f_2)\};$

$T := T \cup \{s'\};$

end

## Checking $EG(f)$

- eliminate (temporarily) all states not fulfilling  $f$
- label all states leading to a non-trivial (i.e. cyclic) strongly connected component (SCC) of the graph representation of the Kripke structure

procedure checkEG( $f$ )

$S' := \{s \mid f \in label(s)\};$

$SCC := \{C \mid C \text{ is non-trivial SCC of } S'\};$       $T := \bigcup_{C \in SCC} C;$

for all  $s \in T$  do  $label(s) := label(s) \cup \{EG f\};$

while  $T \neq \emptyset$  do

    choose  $s \in T$ ;  $T := T - \{s\};$

    for all  $s'$  such that  $s' \in S'$  and  $s' \rightarrow s$  do

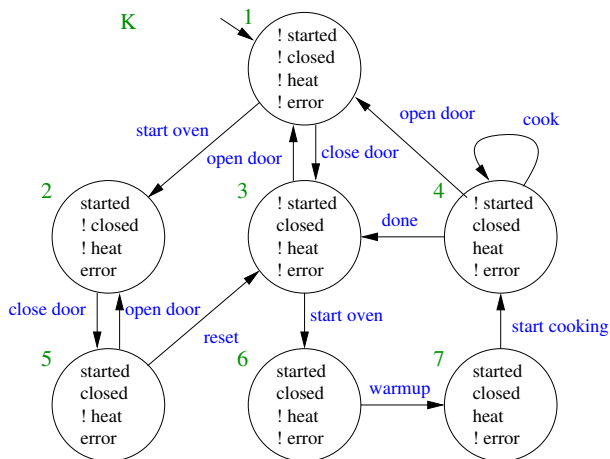
        if  $EG f \notin label(s')$  then

$label(s') := label(s') \cup \{EG f\};$

$T := T \cup \{s'\};$

end

## Example: CTL Model Checking



Is  $f = AG(\text{started} \Rightarrow AF \text{heat})$  fulfilled by  $K$ ?

## Example: CTL Model Checking (2)

- $f = AG(\text{started} \Rightarrow AF \text{heat}) = \neg E(\text{true } U (\text{started} \wedge EG \neg \text{heat}))$
- $S_{\text{started}} = \{2, 5, 6, 7\}$ ,  $S_{\neg \text{heat}} = \{1, 2, 3, 5, 6\}$
- compute  $S_{EG \neg \text{heat}}$  using `checkEG`:
  - $SCC = \{\{1, 2, 3, 5\}\}$ ; no state is added
  - thus:  $S_{EG \neg \text{heat}} = \{1, 2, 3, 5\}$
- $S_{(\text{started} \wedge EG \neg \text{heat})} = \{2, 5\}$  (simplified!  $f_1 \wedge f_2 = \neg(\neg f_1 \vee \neg f_2)$ )
- compute  $S_{E(\text{true } U (\text{started} \wedge EG \neg \text{heat}))}$  using `checkEU`:
  - $T = \{2, 5\}$ , after adding predecessors:
  - $S_{E(\text{true } U (\text{started} \wedge EG \neg \text{heat}))} = \{1, 2, 3, 4, 5, 6, 7\}$
- $S_{\neg E(\text{true } U (\text{started} \wedge EG \neg \text{heat}))} = \emptyset$
- since  $S_0 = \{1\} \not\subseteq \emptyset = S_f$ ,  $f$  is not fulfilled by  $K$ !