Systems and Software Verification

Model-Checking Techniques and Tools

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

Ver. 2.5 (2013.06)

Introduction

- Text
  - System and Software Verification : Model-Checking Techniques and Tools

- In this book, you will find enough theory
  - to be able to assess the relevance of the various tools,
  - to understand the reasons behind their limitations and strengths, and
  - to choose the approach currently best suited for your verification task.

- Part I : Principles and Techniques
- Part II : Specifying with Temporal Logic
- Part III : Some Tools

Chapter 1. Automata

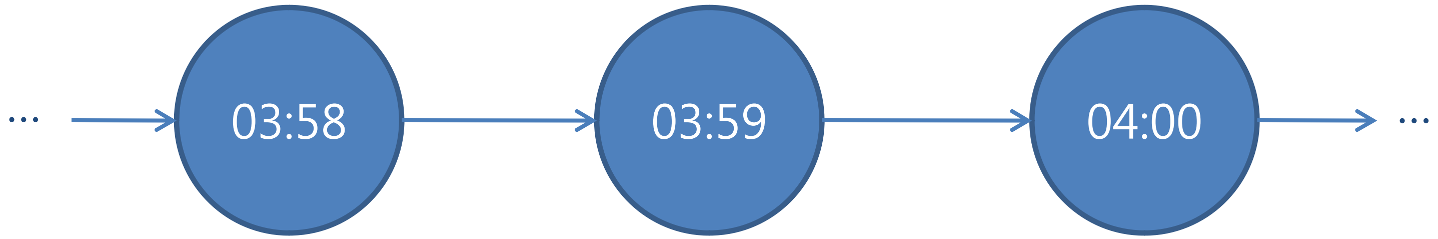
- Model checking consists in verifying some properties of the model of a system.
- Modeling of a system is difficult
  - No universal method exists to model a system
  - Best performed by qualified engineers

- This chapter describes a general model which serves as a basis.

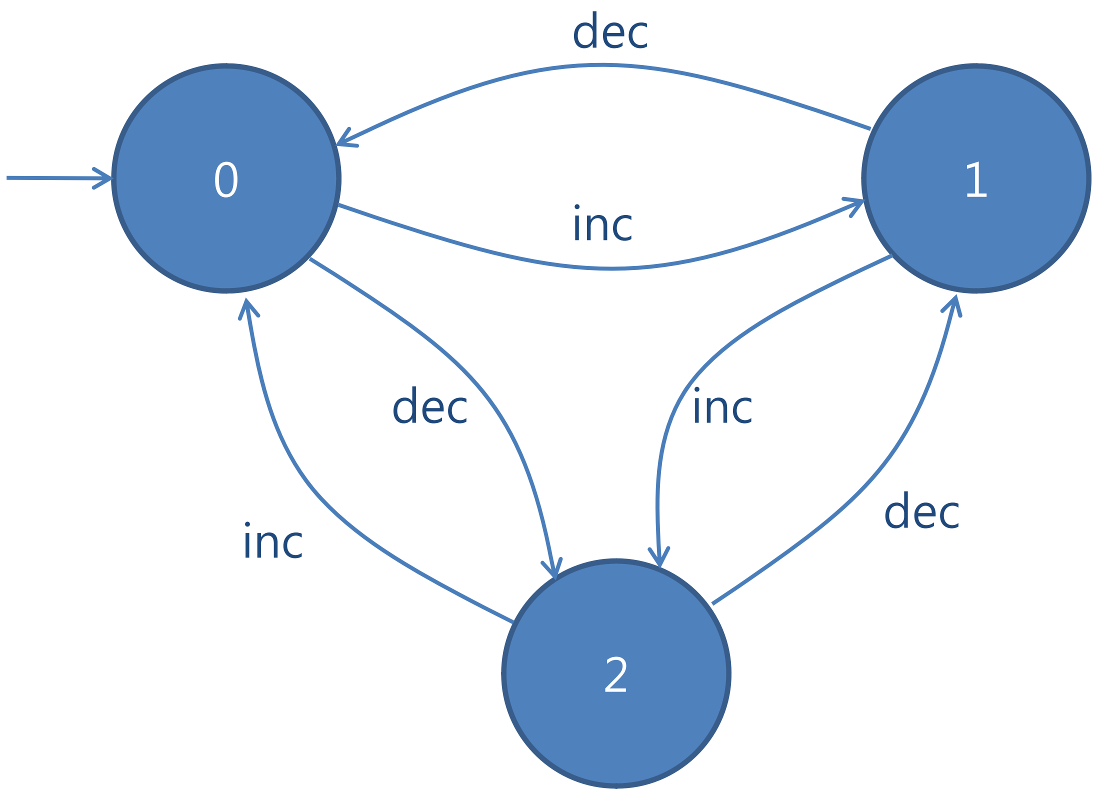
- Organization of Chapter 1
  - Introductory Examples
  - A Few Definitions
  - A Printer Manager
  - A Few More Variables
  - Synchronized Product
  - Synchronization with Messaging Passing
  - Synchronization by Shared Variables

1.1 Introductory Examples

- (Finite) Automata
  - Best suited for verification by model checking techniques
  - A machine evolving from one *state* to another under the action of *transitions*
  - Graphical representation

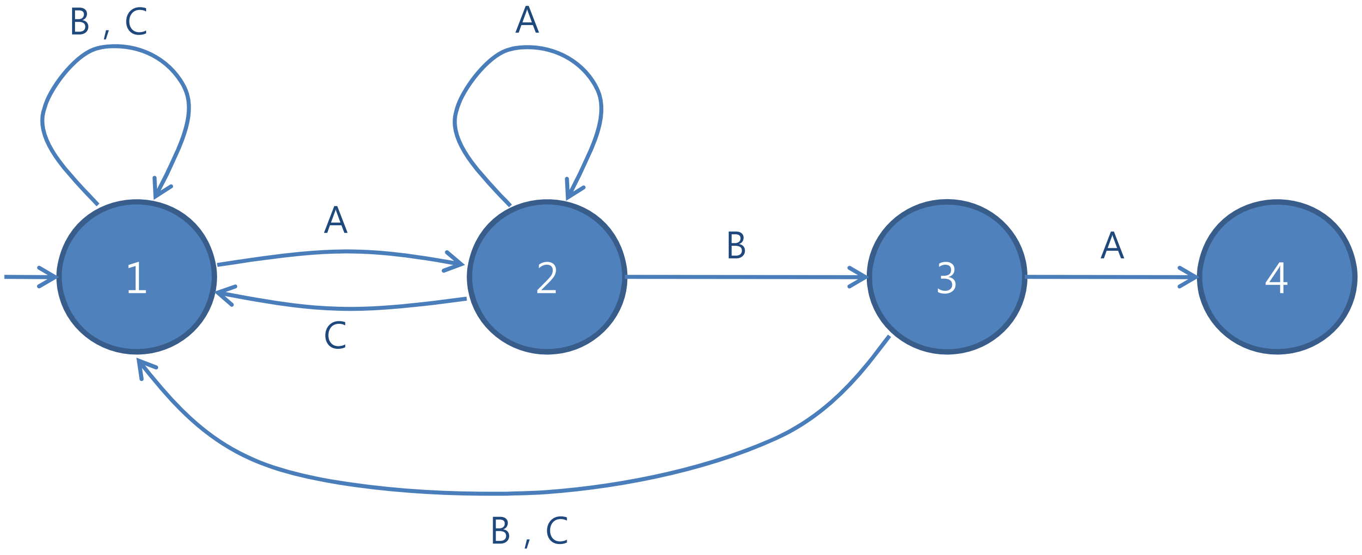


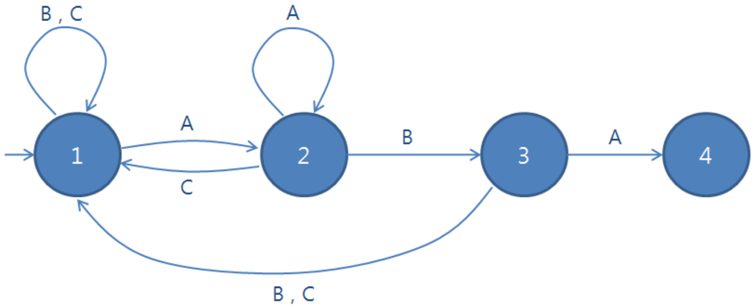
An automate model of a digital watch (24x60=1440 states)



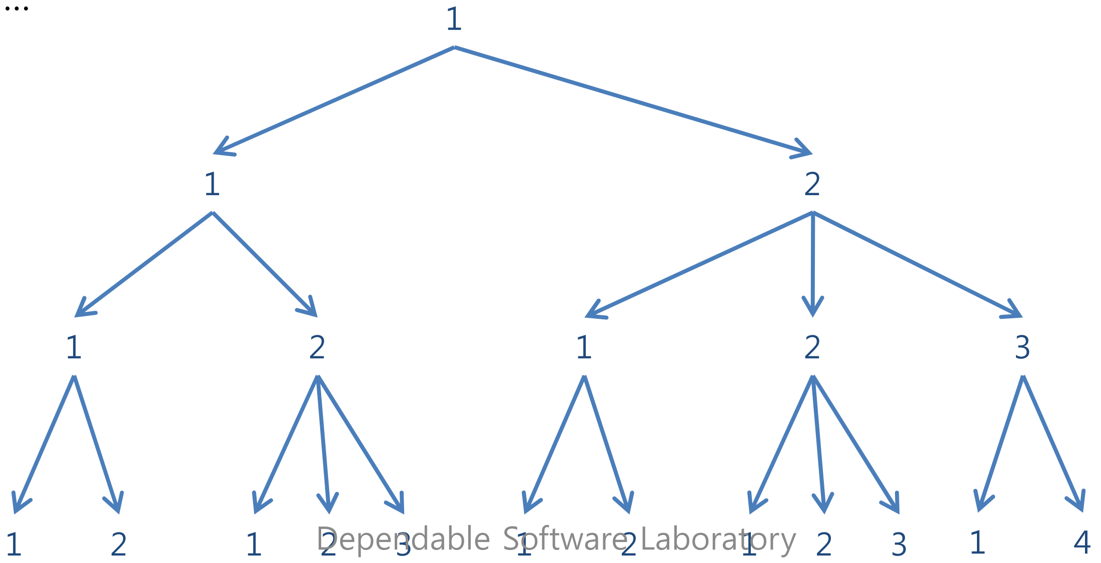
A_{c3} : a module 3 counter

- A digicode door lock example
  - Controls the opening of office doors
  - The door opens upon the keying in of the correct character sequence, irrespective of any possible incorrect initial attempts.
  - Assumes
    - 3 keys A, B, and C
    - Correct key sequence : ABA



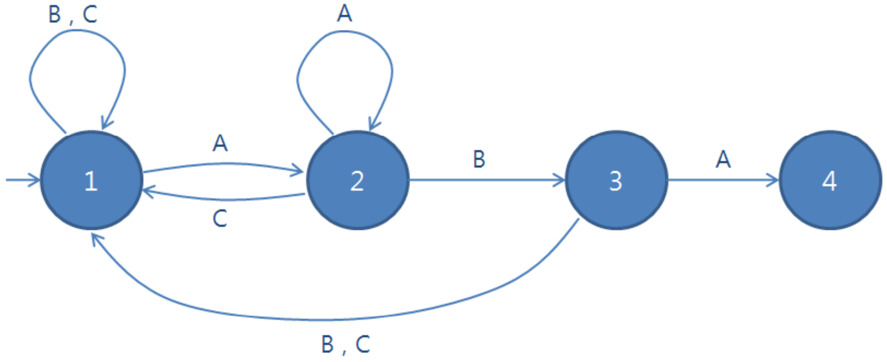


- Two fundamental notations
  - execution
    - A sequence of states describing one possible evolution of the system
    - Ex. 1121 , 12234 , 112312234 ← 3 different executions
  - execution tree
    - A set of all possible executions of the system in the form of a tree
    - Ex. 1
      11, 12
      111, 112, 121, 122, 123
      1111, 1112, 1121, 1122, 1123, 1211, 1212, 1221, 1222, 1223, 1231, 1234
      …



- We associate with each automaton state a number of elementary properties which we know are satisfies, since our goal is to verify system model properties.

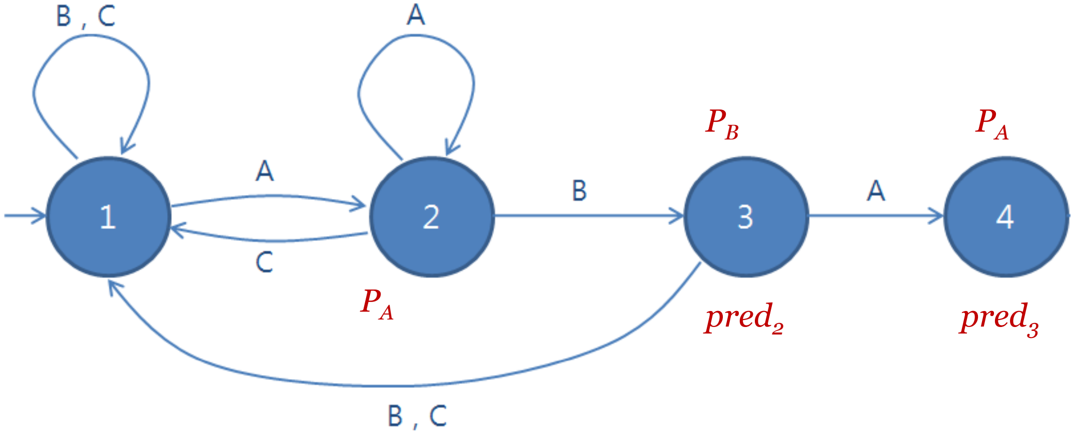
- Properties
  - Elementary property
    - (atomic) Proposition
    - Associated with each state
    - True or False in a given state
  - Complicated property
    - Expressed using elementary properties
    - Depends on the logic we use



- For example,
  - P_A : an A has just been keyed in
  - P_B : an B has just been keyed in
  - P_C : an C has just been keyed in
  - $pred_2$: the proceeding state in an execution is 2
  - $pred_3$: the proceeding state in an execution is 3

  - Properties of the system to verify
    1. If the door opens, then A, B, A were the last three letters keyed in, in that order.
    2. Keying in any sequence of letters ending in ABA opens the door.

  - Let's prove the properties with the propositions



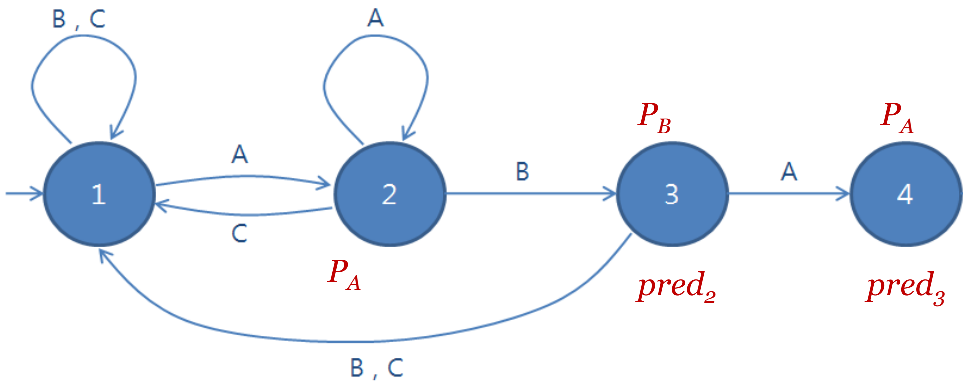
1.2 A Few Definition

- An automaton is a tuple $A = < Q, E, T, q_o, l >$ in which
  - Q : a finite set of states
  - E : the finite set of transition labels
  - $T \subseteq Q \; x \; E \; x \; Q$: the set of transitions
  - q_o : the initial state of the automaton
  - l : the mapping each state with associated sets of properties which hold in it

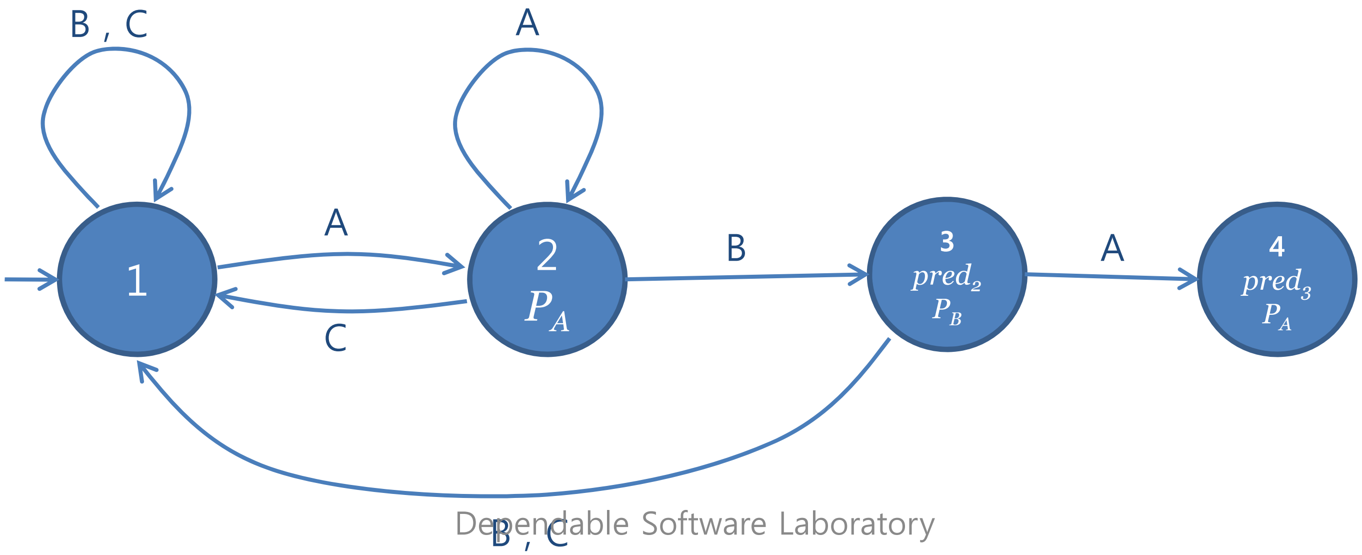
  - $Prop$ = $\{P_{1}, P_{2}, \dots \}$: a set of elementary propositions

$A = < Q, E, T, q_o, l >$

- Q = {1, 2, 3, 4}
- E = $\{A, B, C\}$
- T = { (1,A,2), (1,B,1), (1,C,1), (2,A,2), (2,B,3), (2,C,1), (3,A,4), (3,B,1), (3,C,1) }
- q_o = 1
- $l = \begin{cases} 1 \rightarrow \varnothing \\ 2 \rightarrow \{P_A\} \\ 3 \rightarrow \{P_{B,} pred_2\} \\ 4 \rightarrow \{P_{A,} pred_3\} \end{cases}$

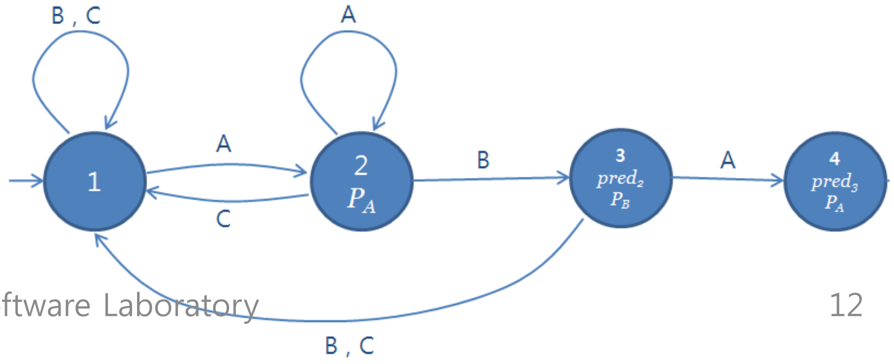


The digicode with its atomic propositions



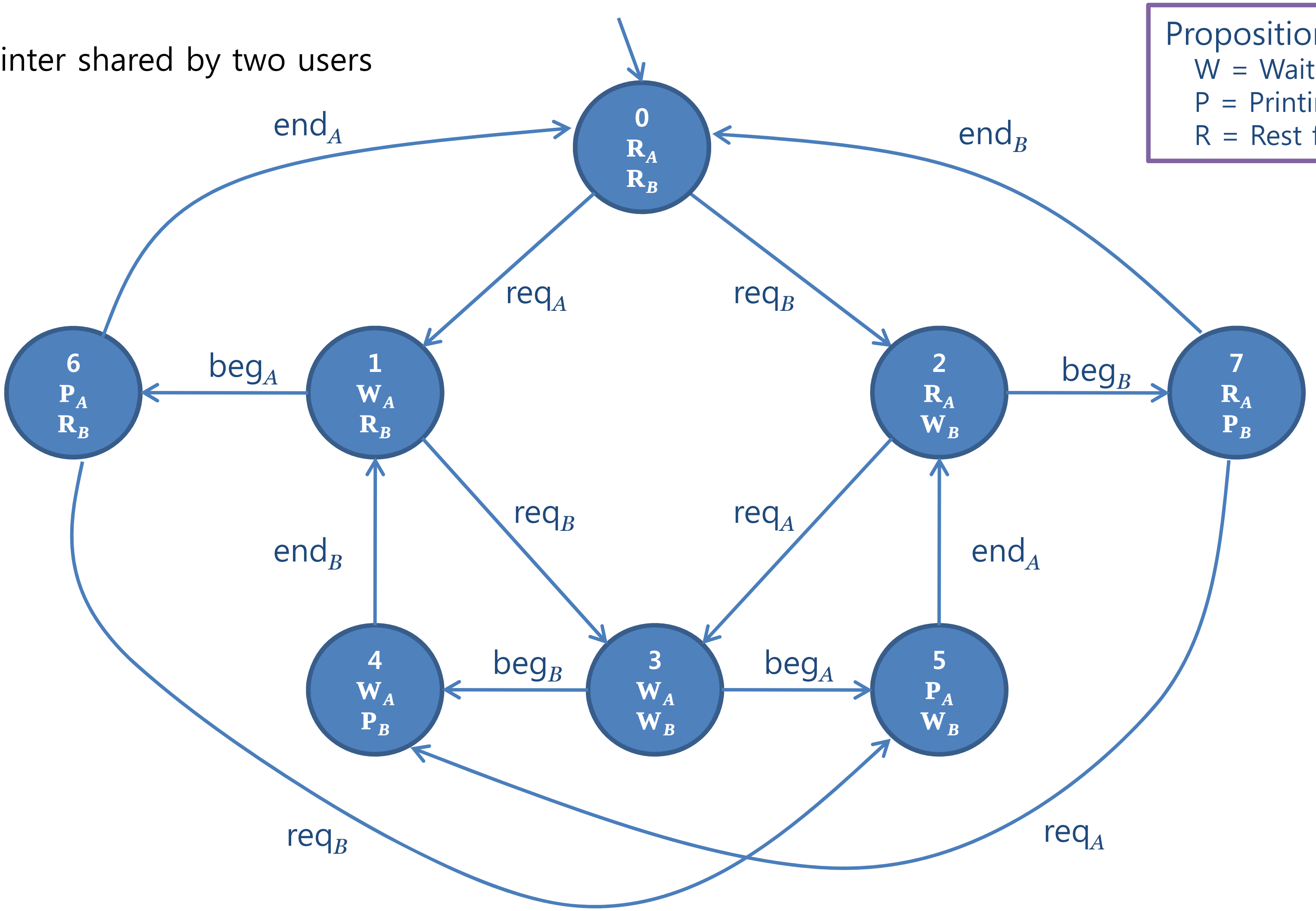
- Formal definitions of automaton's behavior
  - a *path* of automaton A :
    - A sequence σ, finite or infinite, of transitions which follows each other
    - Ex. $3 \xrightarrow{B} 1 \xrightarrow{A} 2 \xrightarrow{A} 2$
  - a *length* of a path σ :
    - $|\sigma|$
    - σ's potentially infinite number of transitions: $|\sigma| \in N \cup \{\omega\}$
  - a *partial execution* of A :
    - A path starting from the initial state q_o
    - Ex. $1 \xrightarrow{A} 2 \xrightarrow{A} 2 \xrightarrow{B} 3$
  - a *complete execution* of A :
    - An execution which is maximal.
    - Infinite or deadlock

  - a *reachable state* :
    - A state is said to be reachable,
      - if a state appears in the execution tree of the automaton, in other words,
      - if there exists at least one execution in which it appears.



1.3 Printer Manager

A printer shared by two users



Propositions

W = Waiting

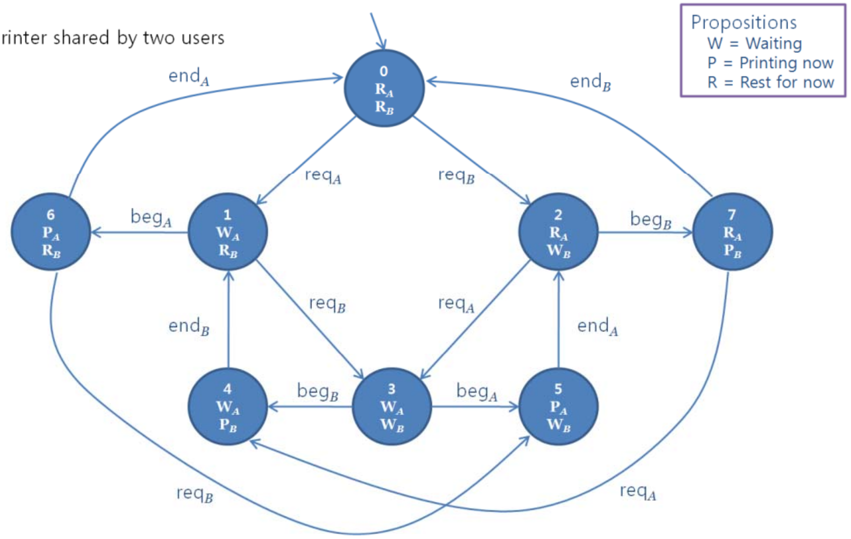
P = Printing now

R = Rest for now

$A = < Q, E, T, q_o, l >$

- Q = {0, 1, 2, 3, 4, 5, 6, 7}
- E = {req$_A$, req$_B$, beg$_A$, beg$_B$, end$_A$, end$_B$}
- T = { (0,req$_A$,1), (0,req$_B$,2), (1,req$_B$,3), (1,beg$_A$,6), (2,req$_A$,3), (2,beg$_B$,7), (3,beg$_A$,5), (3,beg$_B$,4), (4,end$_B$,1), (5,end$_A$,2), (6,end$_A$,0), (6,req$_B$,5), (7,end$_B$,0), (7,req$_A$,4) }
- q_o = 0
- $l = \begin{cases} 0 \rightarrow \{R_A, R_B\}, & 1 \rightarrow \{W_A, R_B\} \\ 2 \rightarrow \{R_A, W_B\}, & 3 \rightarrow \{W_A, W_B\} \\ 4 \rightarrow \{W_A, P_B\}, & 5 \rightarrow \{P_A, W_B\} \\ 6 \rightarrow \{P_A, R_B\}, & 7 \rightarrow \{R_A, P_B\} \end{cases}$

A printer shared by two users



- Properties of the printer manager to verify

1. We would undoubtedly wish to prove that any printing operation is preceded by a print request.
   - In any execution, any state in which P$_A$ holds is preceded by a state in which the proposition W$_A$ holds.

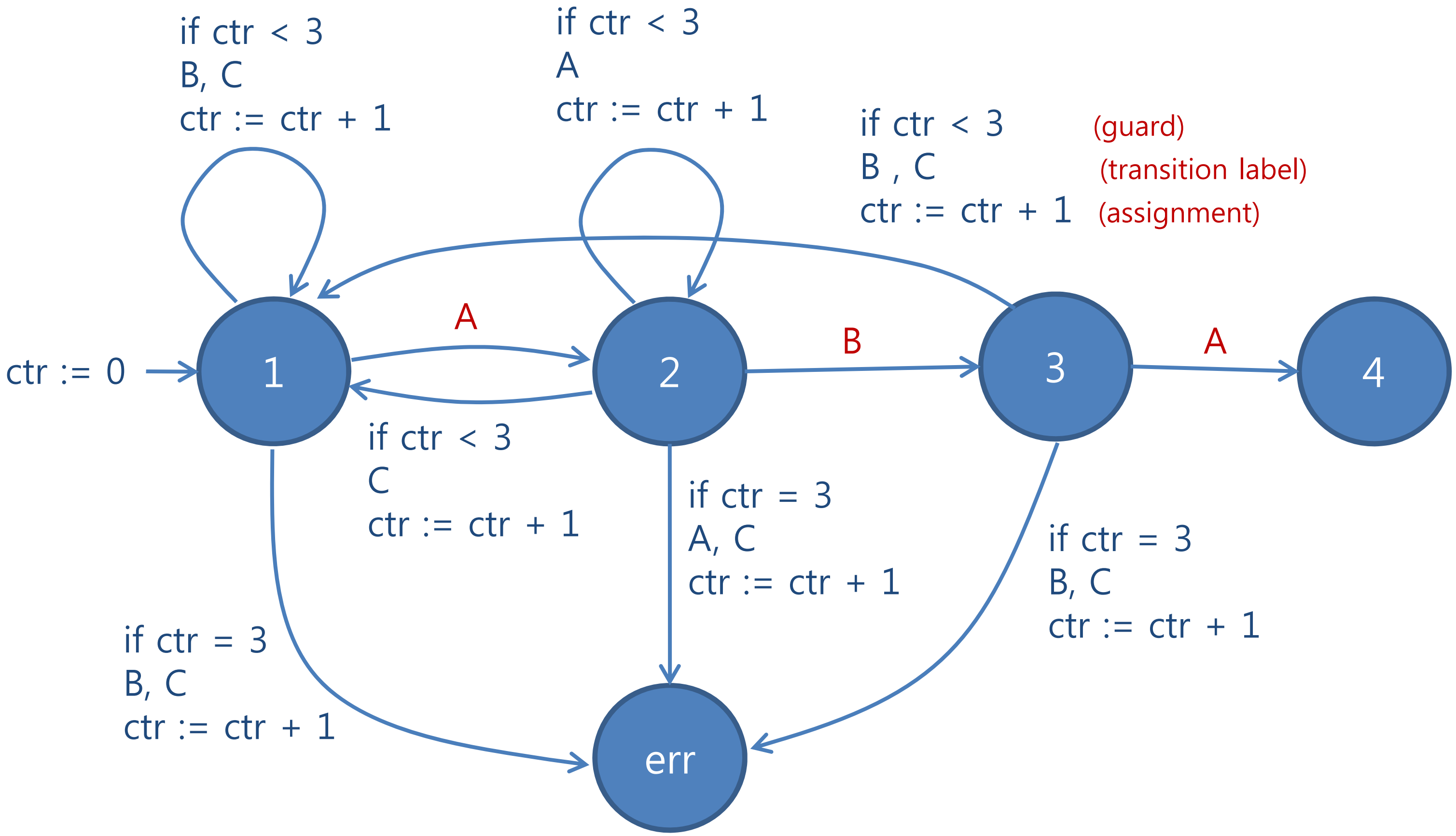
2. Similarly, we would like to check that any print request is ultimately satisfied. (→ fairness property)
   - In any execution, any state in which W$_A$ holds is followed by a state in which the proposition P$_A$ holds.

- Model checking techniques allow us to prove automatically that
  - Property 1 is TRUE, and
  - Property 2 is FALSE, for example 0 1 3 4 1 3 4 1 3 4 1 3 4 1 … (counterexample)

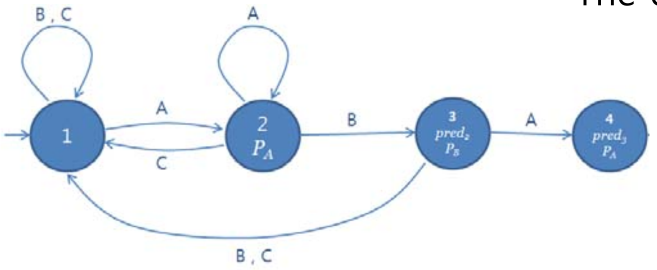
1.4 Few More Variables

- It is often convenient to let automata manipulate *state variables*.
  - Control : states + transitions
  - Data : variables (assumes finite number of values)

- An automaton interacts with variables in two ways:
  - Assignments
  - Guards



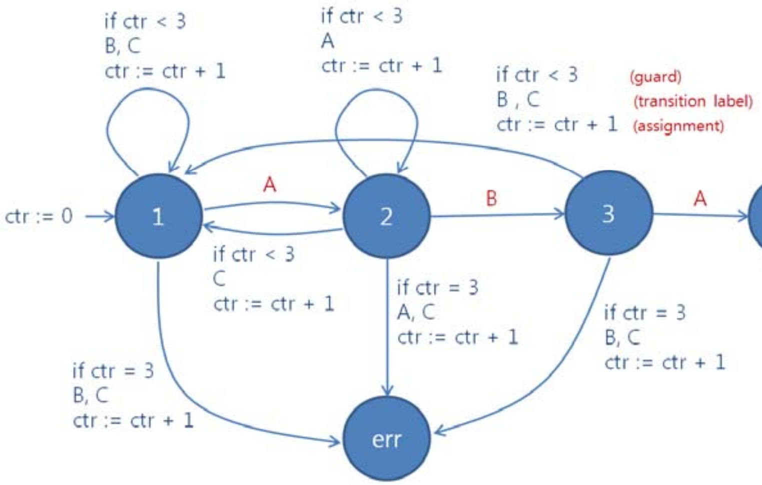
The digicode with guarded transitions



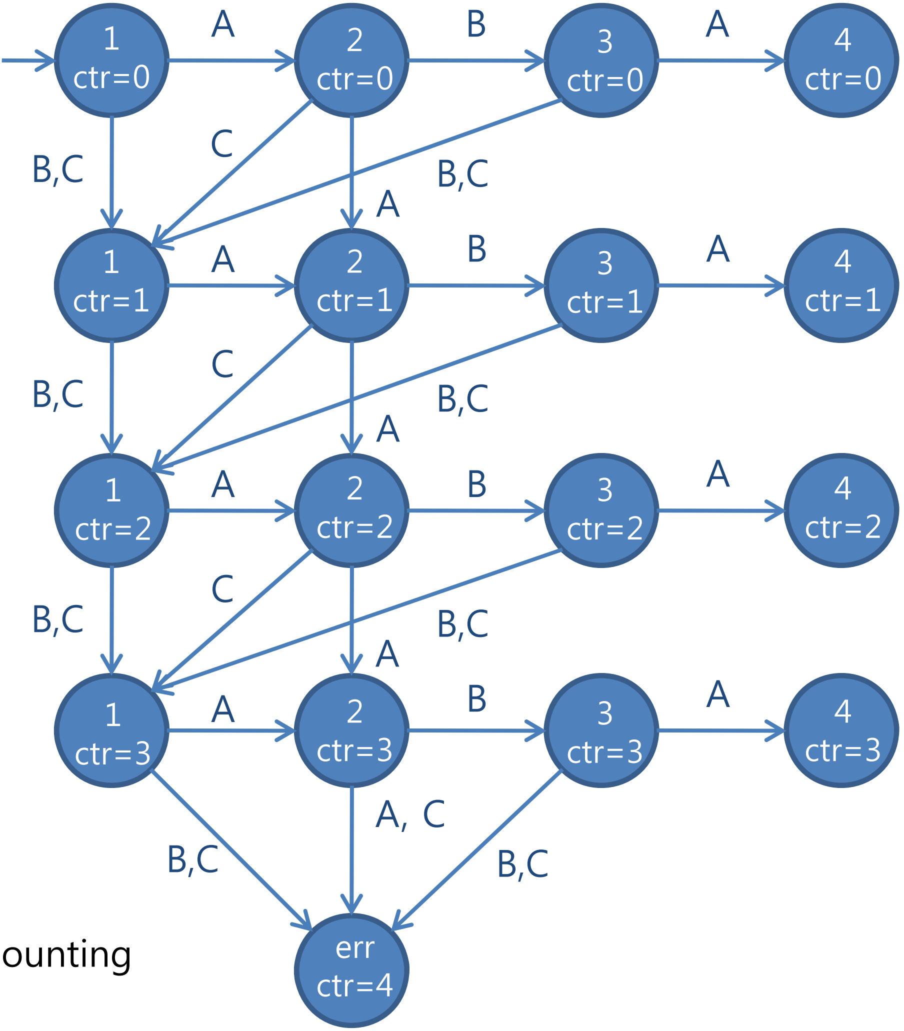
No more than 3 mistakes !!!

- It is often necessary, in order to apply model checking methods,
  - to *unfold* the behaviors of an automaton with variables
  - into a state graph
  - in which the possible transitions appear and the configurations are clear marked.

- Unfolded automaton = Transition system
  - has global states
  - transitions are no longer guarded
  - no assignments on the transitions



Unfolding



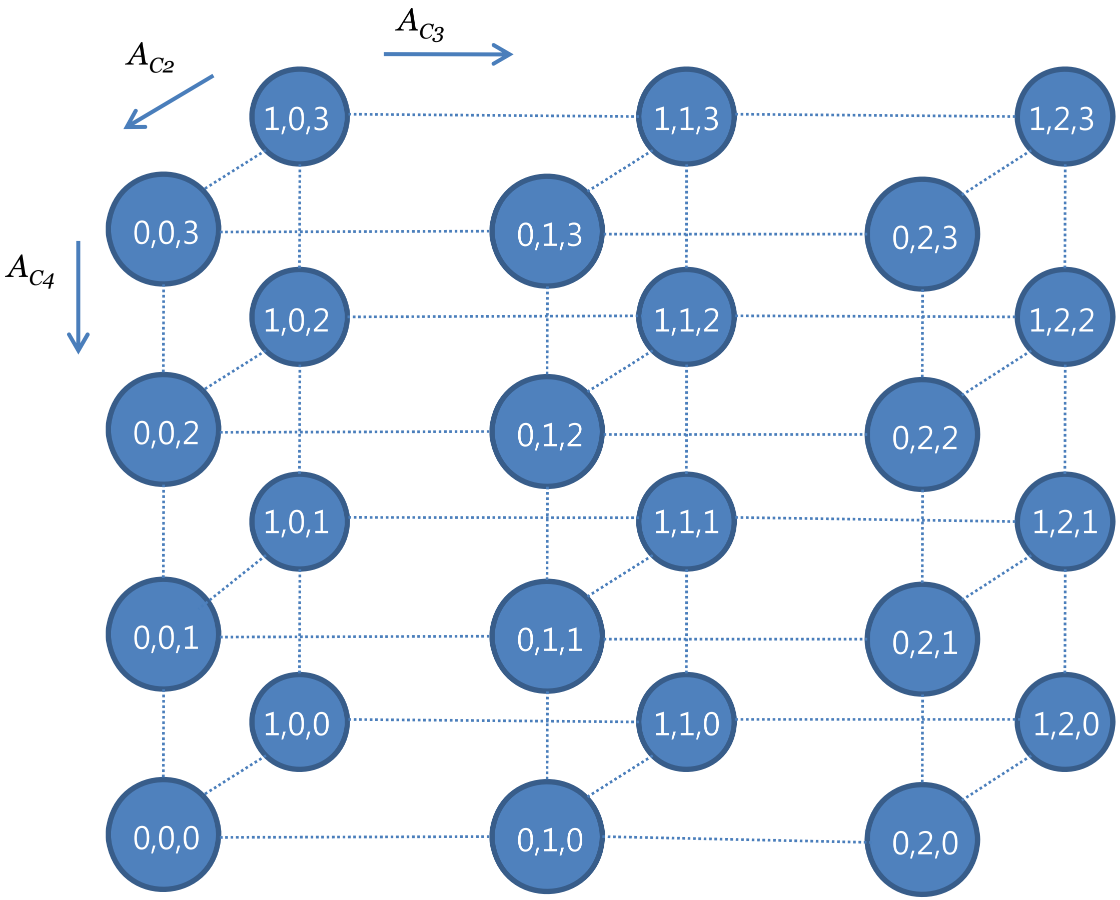
The digicode with error counting
(Unfolded automaton)

1.5 Synchronized Product

- Real-life programs or systems are often composed of modules or subsystems.
  - Modules/Components → (composition) → Overall system
  - Component automata → (synchronization) → Global automaton

- Automata for an overall system
  - Often has so many global states
  - Impossible to construct it directly (State explosion problem)
  - Two composition ways
    - With synchronization
    - Without synchronization

- An example <u>without synchronization</u>
  - A system made up of three counters (modulo 2, 3, 4)
  - They do not interact with each other
  - Global automaton = Cartesian product of three independent automata



2*3*4 = 24 states
3*3*3 - 1 = 26 transitions per a state
(Inc, Dec, -)

→ 24 * 26 = 624 transitions

- An example with synchronization
  - A number of ways depending on the nature of the problem
  - Ex. Allowing only “inc, inc, inc” and “dec, dec, dec” (24*2=48 transitions)
  - Ex. Allowing updates in only one counter at a time (24*3*2=144 transitions)

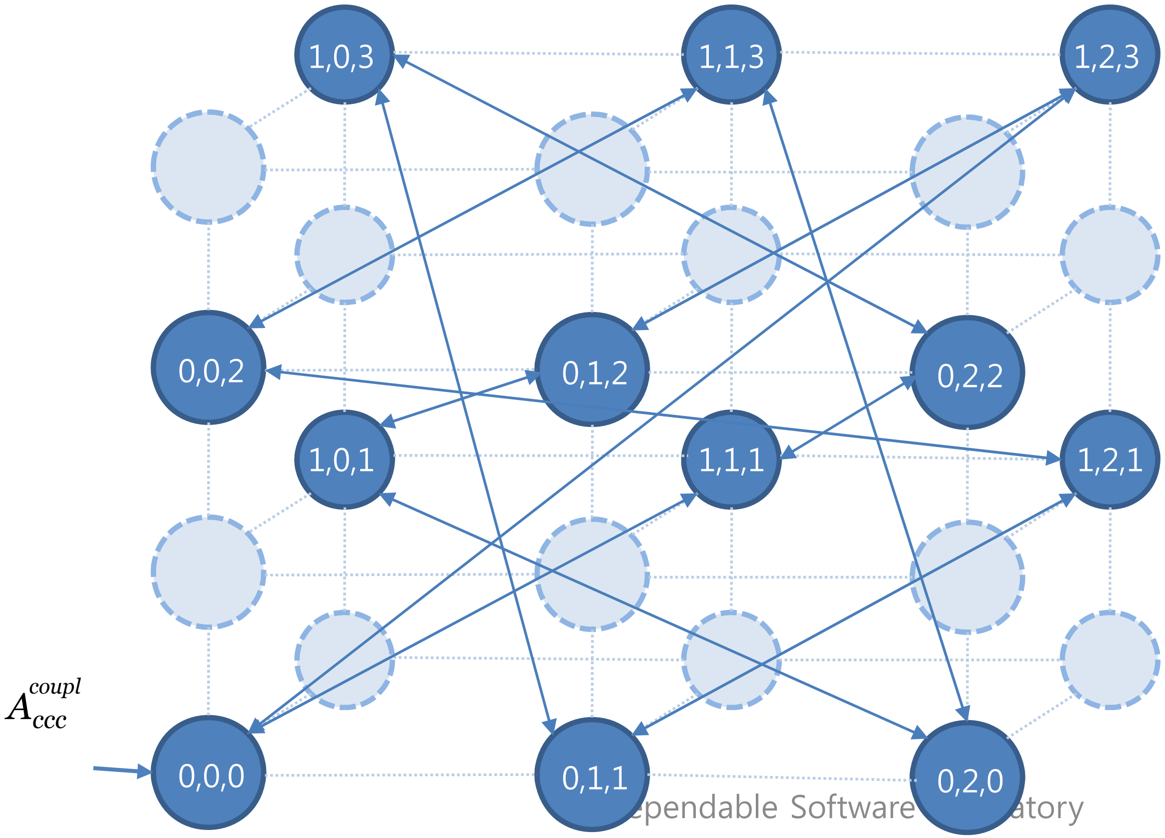
- Synchronized product
  - A way to formally express synchronizing options
  - Synchronized product = Component automata + Synchronized set

  - $A_1 \times A_2 \times \ldots \times A_n$: Component automata

  - $A = < Q, E, T, q_o, l >$
    - $Q = Q_1 \times Q_2 \times \ldots \times Q_n$
    - $E = \prod_{1 \le i \le n} (E_i \cup \{\text{-}\})$
    - $T = \left[\begin{array}{l} ((q_1, \ldots., q_n), (e_1, \ldots, e_n), (q'_1, \ldots, q'_n)) \mid \text{ for all } i, \\ (e_i = \text{‘-’ and } q'_i = q_i) \text{ or } (e_i \neq \text{‘-’ and } (q_i, e_i, q'_i) \in T_i) \end{array} \right]$
    - $q_o = (q_{o,1}, \ldots, q_{o,n})$
    - $l((q_1, \ldots, q_n)) = \bigcup_{1 \le i \le n} l_i (q_i)$

  - $Sync \subseteq \prod_{1 \le i \le n} (E_i \cup \{\text{-}\})$: Synchronized set

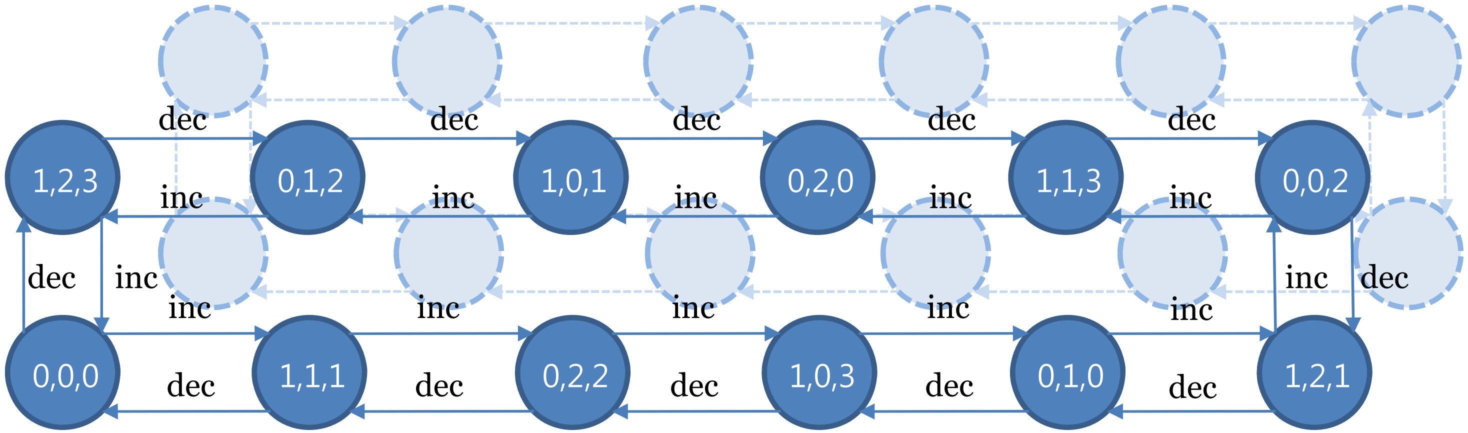
- An example with synchronization
  - Ex. Allowing only “inc, inc, inc” and “dec, dec, dec” (24*2=48 transitions)
    → Strongly coupled version of modular counters
  - *Sync* = { (inc, inc, inc), (dec, dec, dec) }
  - $T = \left\{ \begin{array}{l} ((q_1, \ldots, q_n), (e_1, \ldots, e_n), (q'_1, \ldots, q'_n)) \mid (e_1, \ldots, e_n) \in Sync \\ (e_i = \text{‘-’ and } q'_i = q_i) \text{ or } (e_i \neq \text{‘-’ and } (q_i, e_i, q'_i) \in T_i) \end{array} \right\}$



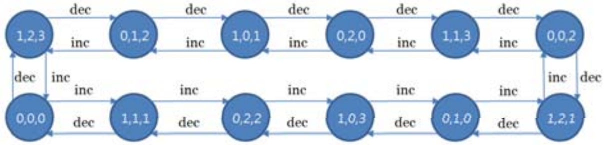
12 states

24 transitions
(inc, inc, inc) (dec, dec, dec)

- Reachable states
  - Reachability depends on the synchronization constraints



Rearranged automaton A_{ccc}^{coupl} → modulo 12 counter



- Reachability graph
  - Obtained by deleting non-reachable states
  - Many tools to construct R.G. of synchronized product of automata
  - Reachability is a difficult problem
    - State explosion problem

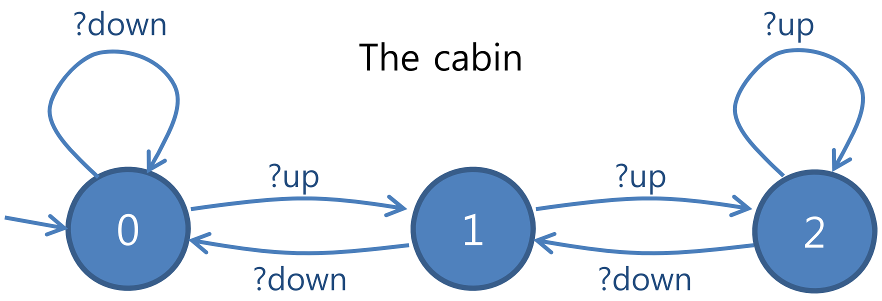
1.6 Synchronization with Message Passing

- Message passing framework
  - A special case of synchronized product
  - !m : Emitting a message
  - ?m : Reception of the message

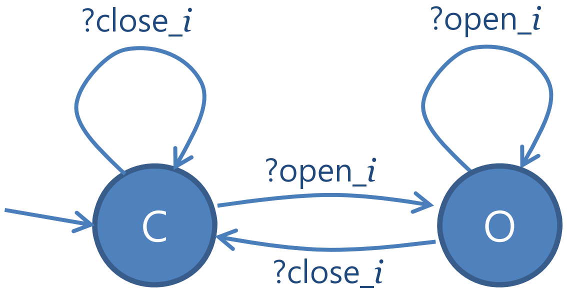
  - Only the transition in which !m and ?m pairs are executed simultaneously is permitted.

  - Synchronous communication
    - Control/command system
  - Asynchronous communication
    - Communication protocol (using channel/buffer)

- Smallish elevator
  - Synchronous communication (message passing)
  - One cabin
  - Three doors (one per floor)
  - One controller
  - No request from the three floors

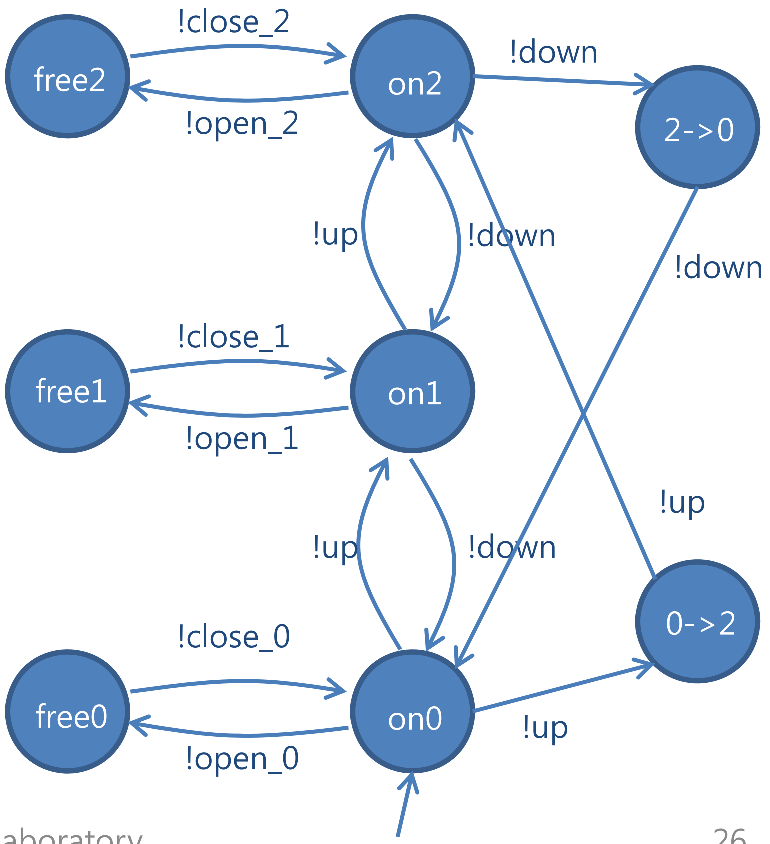


The cabin



The i^{th} door

The controller



- An automaton for the smallish elevator example
  - Obtained as the synchronized product of the five automata
  - (door 0, door 1, door 2, cabin, controller)
  - *Sync* = { (?open_0, -, -, -, !open_0), (?close_0, -, -, -, !close_0),
    (-, ?open_1, -, -, !open_1), (-, ?close_1, -, -, !close_1),
    (-, -, ?open_2, -, !open_2), (-, -, ?close_2, -, !clsoe_2),
    (-, -, -, ?down, !down), (-, -, -, ?up, !up) }

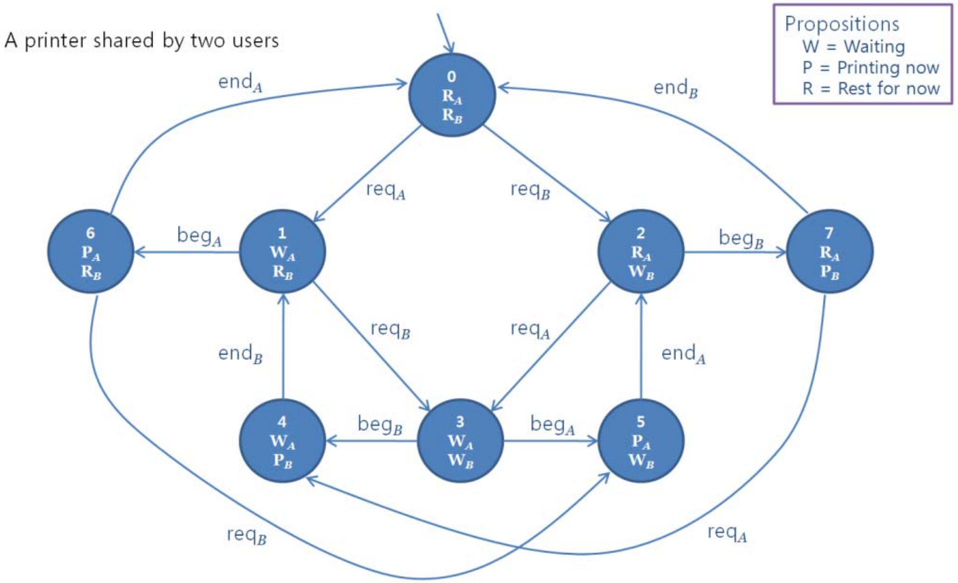
- Properties to check
  - (P1) The door on a given floor cannot open while the cabin is on a different floor.
  - (P2) The cabin cannot move while one of the door is open.

- Model checker
  - Can build the synchronized product of the 5 automata.
  - Can check automatically whether properties hold or not.

1.7 Synchronization by Shared Variables

- Another way to have components communicate with each other
- Share a certain number of variables
- Allow variables to be shared by several automata

- Ex. The printer manager in Chapter 1.3
  - Problem: fairness property is not satisfied

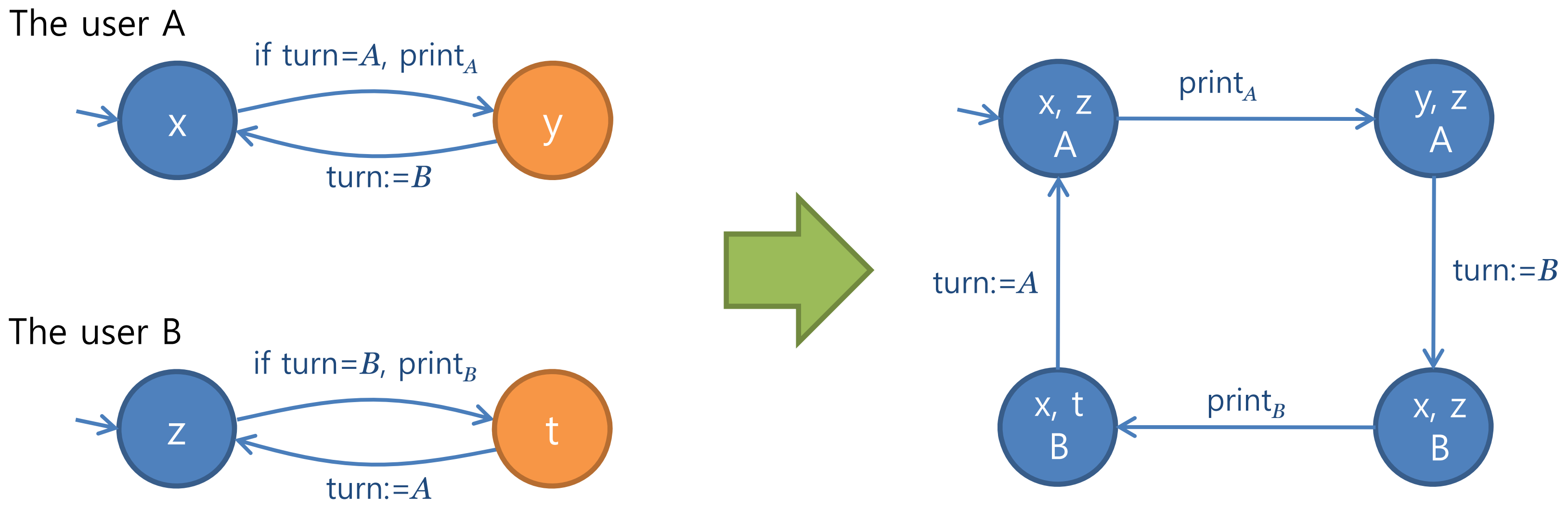


A printer shared by two users

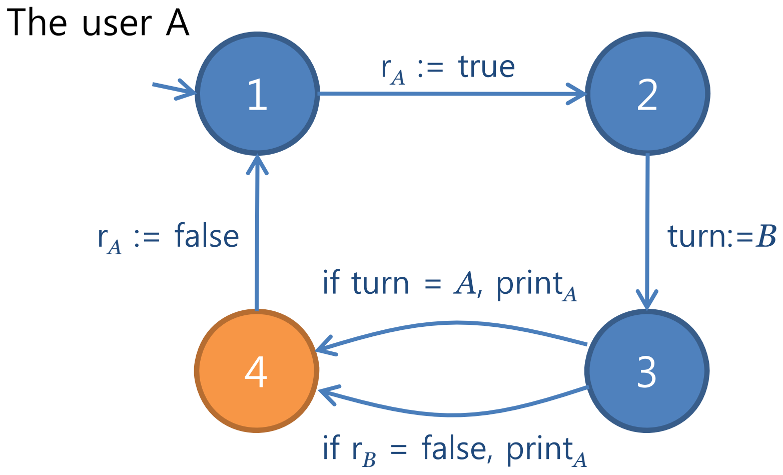
Propositions
W = Waiting
P = Printing now
R = Rest for now

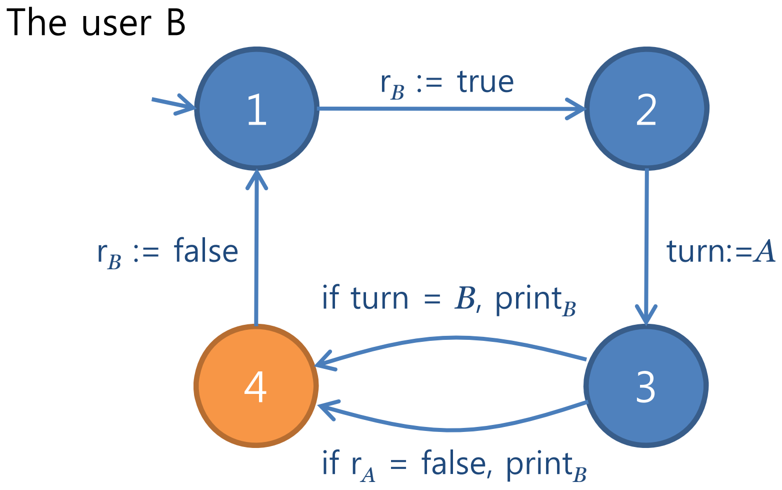
- The printer manager synchronized with a shared variable
  - Shared variable: turn

- Fairness property: “Any print request is ultimately satisfied.”
  → No state of the form (y, t, -) is reachable.
  → TRUE in the model.
  → But, this model forbids either user from printing twice in a row.



- Printer manager : A complete version with 3 variables [by Peterson]
  - r_A : a request from user A
  - r_B : a request from user B
  - turn : to settle conflicts
  - Satisfies all our properties





$A_{A\,x\,B} = < Q, E, T, q_O, l >$

- $Q = A$ x B x r_A x r_B x $turn$
  4 x 4 x 2 x 2 x 2 = 128 states
  (only 128 reachable states)

Chapter 2. Temporal Logic

2. Temporal Logic

- Motivation:
  - The elevator example includes two properties
    - "Any elevator request must ultimately be satisfied"
    - "The elevator never traverses a floor for which a request is pending without satisfying this request"
  - → Dynamic behavior of the system
  - In a first order logic,
    - $\forall t, \forall n \; (app(n, t) \Rightarrow \; \exists t' > t : serv(n, t'))$
    - $\forall t, \forall t' > t, \forall n, \; \begin{bmatrix} (app(n, t) \wedge H(t') \neq n \wedge \; \exists t_{trav} : \\ \quad t \leq t_{trav} \leq t' \leq H(t_{trav}) = n) \\ \quad \Rightarrow (\exists t_{serv} : t \leq t_{serv} \leq t' \wedge serv(n, t_{serv})) \end{bmatrix}$
  - But, the above notation(mathematics) is quite cumbersome.
- Temporal Logic is a different formalism, better suited for our situation.

2. Temporal Logic

- Temporal Logic
  - A form of logic specifically tailored for
    - statements and reasoning
    - involving the notion of order in time
  - Compared with the mathematical formulas
    - clearer and simpler
    - immediately ready for use (linguistic similarity of operators)
    - formal semantics (specification language tools)

- Organization of Chapter 2
  - The Language of Temporal Logic
  - The Formal Syntax of Temporal Logic
  - The Semantics of Temporal Logic
  - PLTL and CTL: Two Temporal Logics
  - The Expressivity of CTL*

2.1 The Language of Temporal Logic

- CTL*
  - serves to formally state the properties concerned with the execution of a system
  - variants (CTL, PLTL, LTL)
  - 6 characteristics

1. Atomic Propositions

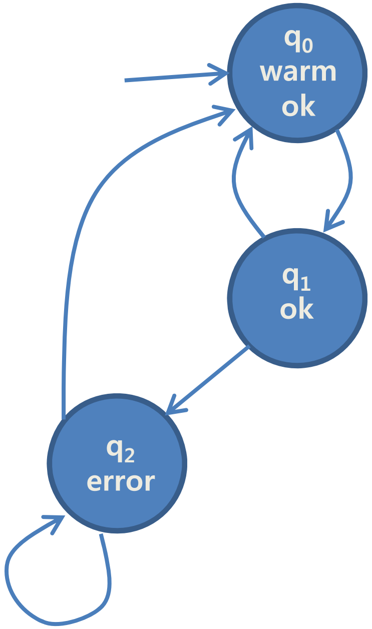
   - *warm, ok, error*

2. Proposition Formula

   - using boolean combinators
   - true, false, ¬, ∨, ∧, ⇒ (if then), ⇔ (if and only if)

   - $error \Rightarrow \neg\, warm$

     (if $error$ then not $warm$)



σ_1 : (q_0: warm, ok) → (q_1: ok) → (q_0: warm, ok) → (q_1: ok) → …

σ_2 : (q_0: warm, ok) → (q_1: ok) → (q_2: error) → (q_0: warm, ok) →
(q_1: ok) → …

σ_3 : (q_0: warm, ok) → (q_1: ok) → (q_2: error) → (q_2: error) →
(q_2: error) → …

3. Temporal combinators

- about the sequencing of states along an execution

- X : next state
- F : a future state
- G : all the future states

- X P : the next state satisfies P
- F P : a future state satisfies P without specifying which state
  → P will hold some day (at least once)
- G P : all future states will satisfy P
  → P will always be

- $alert \Rightarrow$ F $halt$: if we are currently in a state of $alert$, then we will later be in a $halt$ state.
- G ($alert \Rightarrow$ F $halt$) : at any time, a state of $alert$ will necessarily be followed by a $halt$ state later.

- G ($warm \Rightarrow$ F $\neg warm$) : true
- G ($warm \Rightarrow$ X $\neg warm$) : true

- G is the dual of F
  - G $\phi \equiv \neg$ F$\neg\phi$

4. Arbitrary nesting of temporal combinators

- giving temporal logic its power and strength

- GF ϕ : always there will some day be a state such that ϕ,
  ϕ is satisfied infinitely often along the execution considered
- FG ϕ : all the time from a certain time onward, at each time instant,
  possibly excluding a finite number of instants

- GF $warm \vee$ FG $error$

5. U combinator

- for until
- ϕ_1 U ϕ_2 : ϕ_1 is verified until ϕ_2 is verified
  ϕ_2 will be verified some day, and ϕ_1 will hold in the meantime

- G ($alert \Rightarrow$ ($alarm$ U $halt$)) : starting from a state of *alert*, the *alarm* remains activated
  until the *halt* state is eventually and inexorably reached.

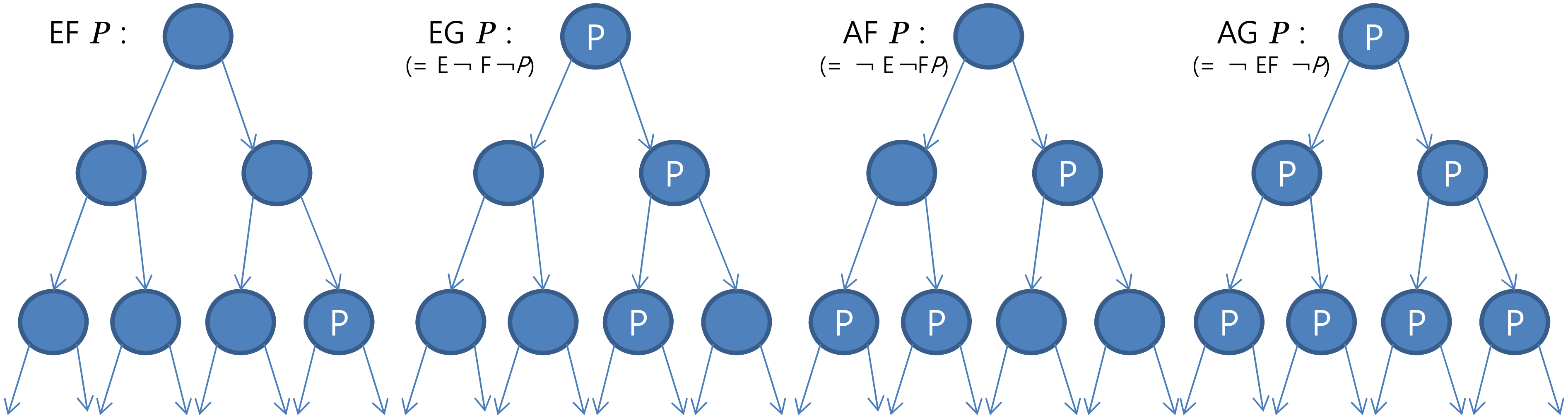
- F $\phi \equiv true$ U ϕ
- ϕ_1 W $\phi_2 \equiv$ (ϕ_1 U ϕ_2) $\vee$ G ϕ_1 : weak until

6. Path quantifier

- A ϕ : all the executions out of the current state satisfy property ϕ
- E ϕ : from the current state, there exists an execution satisfying ϕ

- EF P : it is possible (by following a suitable execution) to have P some day
- EG P : there exists an execution along which P always holds

- AF P : we will necessarily have P some day (regardless of the chosen execution)
- AG P : always true



EF P :

EG P :
(= E¬ F¬P)

AF P :
(= ¬ E¬FP)

AG P :
(= ¬ EF ¬P)

2.2 Formal Syntax of Temporal Logic

- Abstract grammar
  - needs parentheses, operator priority, specific set of atomic propositions, etc.
  - Most model checkers use a fragment of CTL* - CTL or LTL.

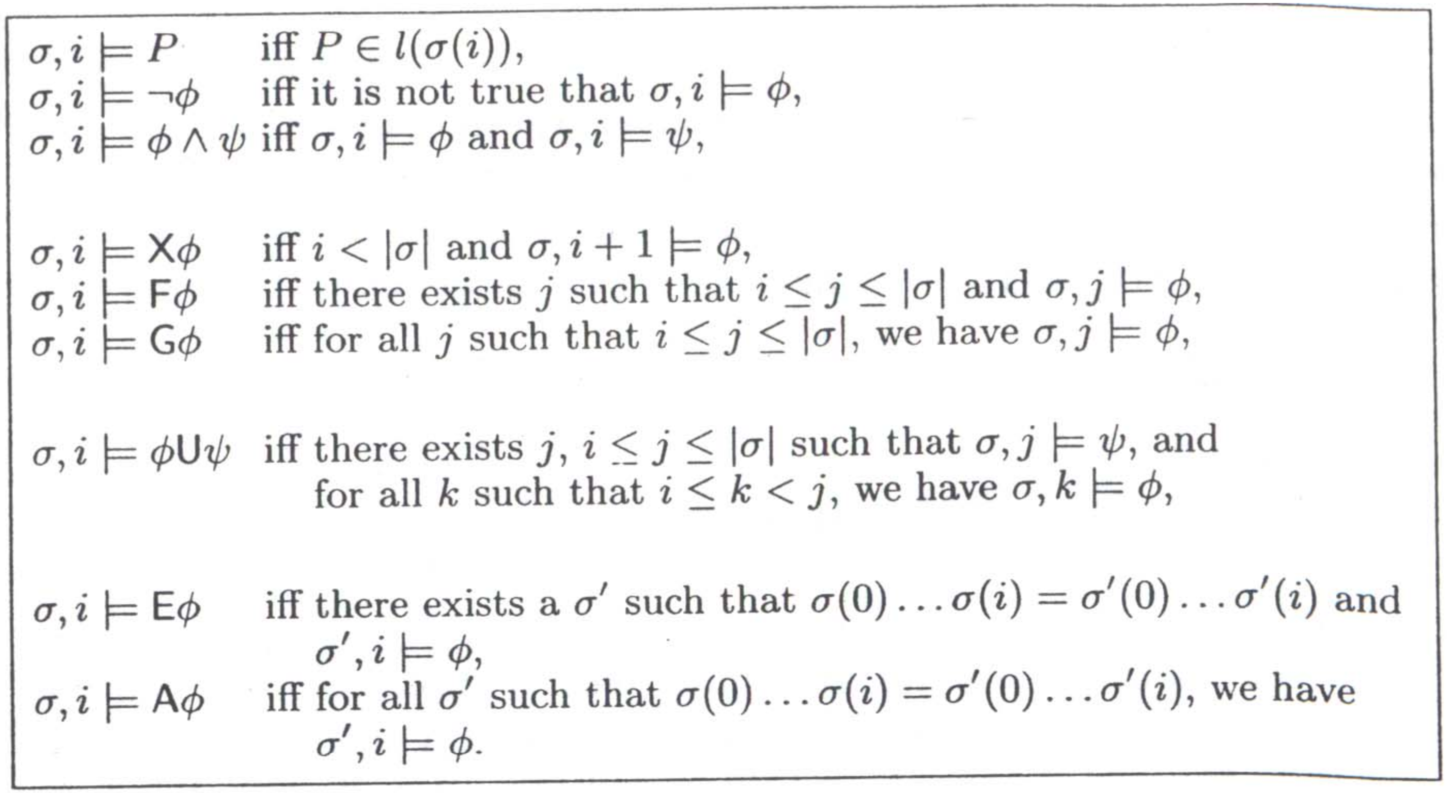
$$\begin{aligned} \phi, \Psi ::= {} & P_1 \mid P_2 \mid \dots && \text{(atomic proposition)} \\ & \mid \neg\phi \mid \phi \wedge \Psi \mid \phi \Rightarrow \Psi \mid \dots && \text{(boolean combinators)} \\ & \mid \mathrm{X}\phi \mid \mathrm{F}\phi \mid \mathrm{G}\phi \mid \phi\, \mathrm{U}\, \Psi \mid \dots && \text{(temporal combinators)} \\ & \mid \mathrm{E}\phi \mid \mathrm{A}\phi && \text{(path quantifiers)} \end{aligned}$$

2.3 The Semantics of Temporal Logic

- Kripke structure
  - Name of the models of temporal logic
  - Propositions labeling the states are important in CTL*
  - Transition labels (E) are neglected. $A = < Q, T, q_o , l >$, $T \subseteq Q \times Q$

- Satisfaction
  - $A,\sigma,i \models \phi$
    - "at time i of the execution σ, ϕ is true."
    - where σ is an execution of A, which not required to start at the initial state
    - A is often omitted.
  - $\sigma,i \models \phi$: ϕ is satisfied at time i of σ
  - $\sigma,i \not\models \phi$: ϕ is not satisfied at time i of σ

  - $A \models \phi$ iff σ,0 $\models \Phi$ for every execution of σ of A
    - "the automaton A satisfies ϕ"
    - $A \not\models \phi \neq A \models \neg\phi$
    - $\sigma,i \not\models \phi = \sigma,i \models \neg\phi$



$\sigma, i \models P$ iff $P \in l(\sigma(i))$,

$\sigma, i \models \neg\phi$ iff it is not true that $\sigma, i \models \phi$,

$\sigma, i \models \phi \wedge \psi$ iff $\sigma, i \models \phi$ and $\sigma, i \models \psi$,

$\sigma, i \models \mathsf{X}\phi$ iff $i < |\sigma|$ and $\sigma, i+1 \models \phi$,

$\sigma, i \models \mathsf{F}\phi$ iff there exists j such that $i \leq j \leq |\sigma|$ and $\sigma, j \models \phi$,

$\sigma, i \models \mathsf{G}\phi$ iff for all j such that $i \leq j \leq |\sigma|$, we have $\sigma, j \models \phi$,

$\sigma, i \models \phi\mathsf{U}\psi$ iff there exists j, $i \leq j \leq |\sigma|$ such that $\sigma, j \models \psi$, and for all k such that $i \leq k < j$, we have $\sigma, k \models \phi$,

$\sigma, i \models \mathsf{E}\phi$ iff there exists a σ' such that $\sigma(0)\ldots\sigma(i) = \sigma'(0)\ldots\sigma'(i)$ and $\sigma', i \models \phi$,

$\sigma, i \models \mathsf{A}\phi$ iff for all σ' such that $\sigma(0)\ldots\sigma(i) = \sigma'(0)\ldots\sigma'(i)$, we have $\sigma', i \models \phi$.

Semantics of CTL*

CTL*

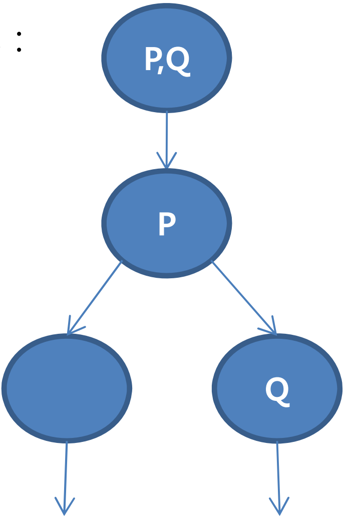
- Time is discrete.
- Nothing exists between i and i + 1.
- The instants are the points along the executions

2.4 PLTL and CTL: Two Temporal Logics

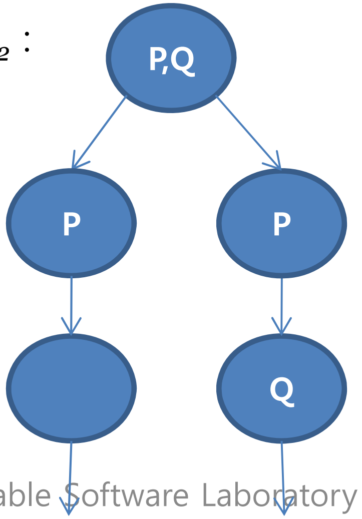
- Two most commonly used temporal logics in model checking tools
  - PLTL (Propositional Linear Temporal Logic)
  - CTL (Computational Tree Logic)
  - fragments of CTL*

- PLTL
  - No path quantifiers (A and E)
  - Linear time logic → Path formula
  - For example, PLTL cannot distinguish A_1 from A_2

A_1 :

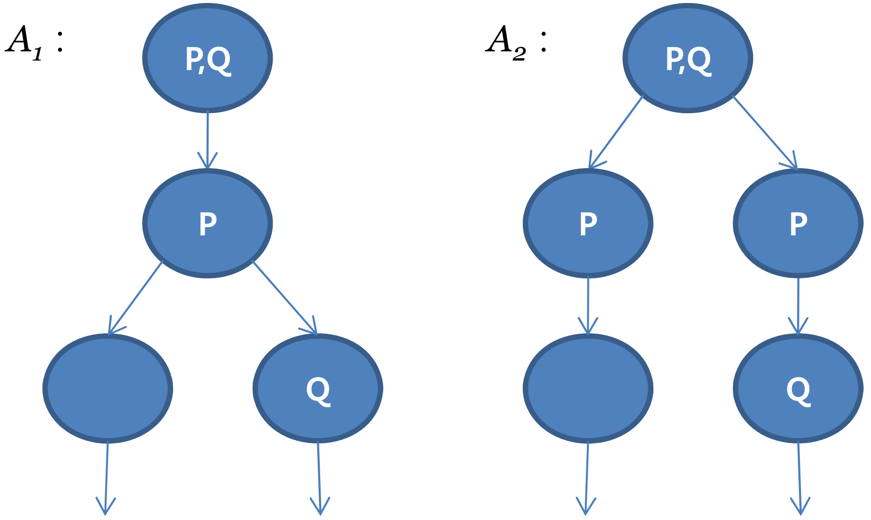


A_2 :



Execution 1 : {P, Q} . {P}. {-}
Execution 2 : {P, Q} . {P} . {Q}

- CTL
  - Temporal combinators (X, F, U) should be under the immediate scope of path quantifier (A, E)
  - EX , AX , EU , AU , EF , EG , AG , AF , …
  - State formulas
    - Truth only depends on the current state and the automaton regions made reachable by it
    - not depending on a current execution
    - $q \models \phi$: ϕ is satisfied in state q
    - CTL can distinguish automata $A1$ and $A2$



$A_1, q_0 \models$ AX (EXQ $\wedge$ EX$\neg Q$)
$A_2, q'_0 \not\models$ AX (EXQ $\wedge$ EX$\neg Q$)

  - Potential reachability : AG EF P
  - Do not allow to express very rich properties along the paths.

- Which to choose CTL or PLTL ?
  - To state some properties
    → PLTL
  - To perform exhaustive verification of a system
    → CTL
  - For both purposes
    → CTL*
    - Less popular
    - More complicated than PLTL
  - CTL + Fairness properties → FCTL
  - If we use model checking tools, then we have no choice
    - SMV : CTL (CTL*)
    - SPIN : PLTL
    - VIS : CTL / PLTL

2.5 The Expressivity of CTL*

- No logic can express anything not taken into account by the modeling decision made

- CTL* is rather expressive enough, when
  - Properties concern the execution tree of our automata
  - CTL* combinators are sufficiently expressive
  - CTL* is almost always sufficient

Chapter 3. Model Checking

3. Model Checking

- Motivation:
  - Describe the principles underlying the algorithms used for model checking
  - The algorithm
    - Can find out whether a given automaton satisfies a given temporal formula
    - Different algorithms for CTL and PLTL

- Organization of Chapter 3
  - Model Checking CTL
  - Model Checking PLTL
  - The State Explosion Problem

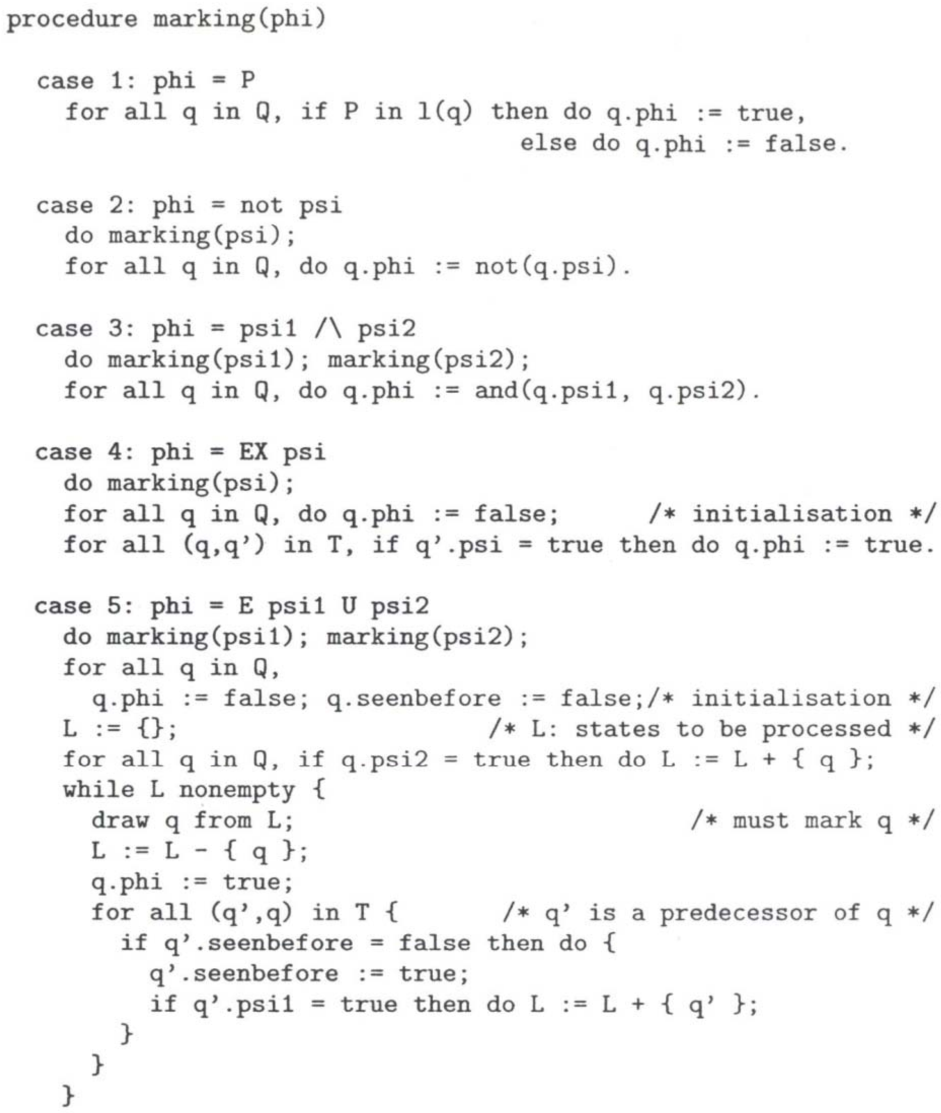
3.1 Model Checking CTL

- Model checking algorithm for CTL
  - Developed in 1980s
  - Runs in time linear in each of its components (automaton and CTL formula)
  - Relies on the fact that CTL can only express state formulas

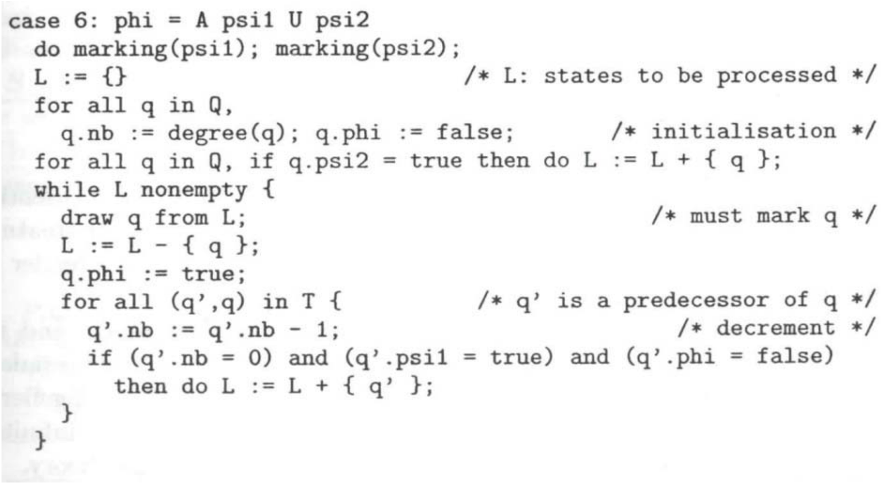
- Basic principles
  - procedure **marking**
    - Starting from a CTL formula ϕ
    - Mark for each state q of the automaton and for each sub-formula ψ of ϕ,
    - Whether ψ is satisfied in state q

- Correctness of the algorithm
  - …
  - Hence, the marking of q is correct.

- Complexity of the algorithm
  - Model checking " does $A,q_o \models \Phi$? " for a CTL formula ϕ
  - can be solved in time O($|A|$ x $|\phi|$)
    - O($|A|$) : for marking the automaton
    - O($|\phi|$) : for each sub-formula in ϕ
  - Linear!!!



```
procedure marking(phi)

  case 1: phi = P
    for all q in Q, if P in l(q) then do q.phi := true,
                                  else do q.phi := false.

  case 2: phi = not psi
    do marking(psi);
    for all q in Q, do q.phi := not(q.psi).

  case 3: phi = psi1 /\ psi2
    do marking(psi1); marking(psi2);
    for all q in Q, do q.phi := and(q.psi1, q.psi2).

  case 4: phi = EX psi
    do marking(psi);
    for all q in Q, do q.phi := false;       /* initialisation */
    for all (q,q') in T, if q'.psi = true then do q.phi := true.

  case 5: phi = E psi1 U psi2
    do marking(psi1); marking(psi2);
    for all q in Q,
      q.phi := false; q.seenbefore := false;/* initialisation */
    L := {};                       /* L: states to be processed */
    for all q in Q, if q.psi2 = true then do L := L + { q };
    while L nonempty {
      draw q from L;                               /* must mark q */
      L := L - { q };
      q.phi := true;
      for all (q',q) in T {        /* q' is a predecessor of q */
        if q'.seenbefore = false then do {
          q'.seenbefore := true;
          if q'.psi1 = true then do L := L + { q' };
        }
      }
    }
```



```
case 6: phi = A psi1 U psi2
  do marking(psi1); marking(psi2);
  L := {}                              /* L: states to be processed */
  for all q in Q,
    q.nb := degree(q); q.phi := false;       /* initialisation */
  for all q in Q, if q.psi2 = true then do L := L + { q };
  while L nonempty {
    draw q from L;                                   /* must mark q */
    L := L - { q };
    q.phi := true;
    for all (q',q) in T {             /* q' is a predecessor of q */
      q'.nb := q'.nb - 1;                              /* decrement */
      if (q'.nb = 0) and (q'.psi1 = true) and (q'.phi = false)
        then do L := L + { q' };
    }
  }
```

3.2 Model Checking PLTL

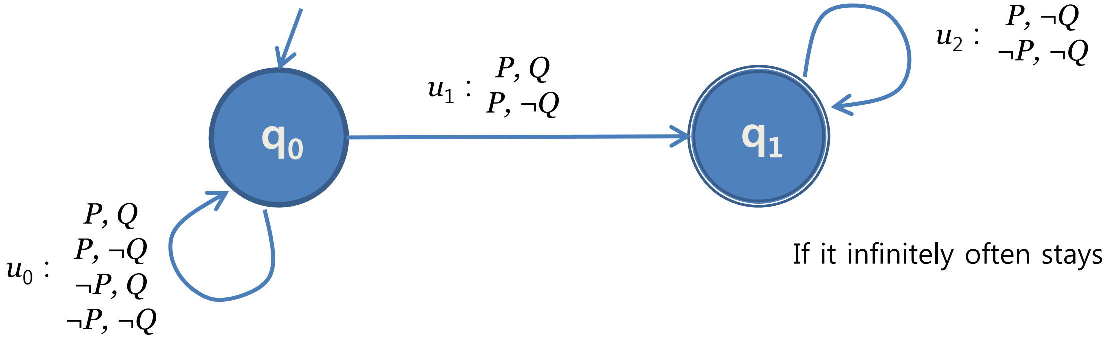
- Model checking algorithm for PLTL
  - Developed in 1980s, but too technical to cover in this course
  - PLTL uses path formulas
  - No longer possible to rely on marking the automaton states
  - A finite automaton will generally give rise to infinitely many different executions, themselves often infinite in length
  - Hence, PLTL uses a language theory : ω-regular expression
    - An extension of a regular expression
    - "*" : an arbitrary but finite number of repetitions
      - (a b* + c)*
    - "ω": an infinite number of repetitions

- Basic principle
  - Model checking " does $A \models \phi$? " for a PLTL formula ϕ
  - Reduces to a " Are all the execution of A described by ε_{ϕ} ? "

  - A PLTL model checker construct an automaton $B_{\neg\phi}$ (recognizing executions which do not satisfy ϕ)
  - Strongly synchronize A and $B_{\neg\phi}$ → $A \otimes B_{\neg\phi}$

  - Finally reduces to " Is the language recognized by $A \otimes B_{\neg\phi}$ empty ?"

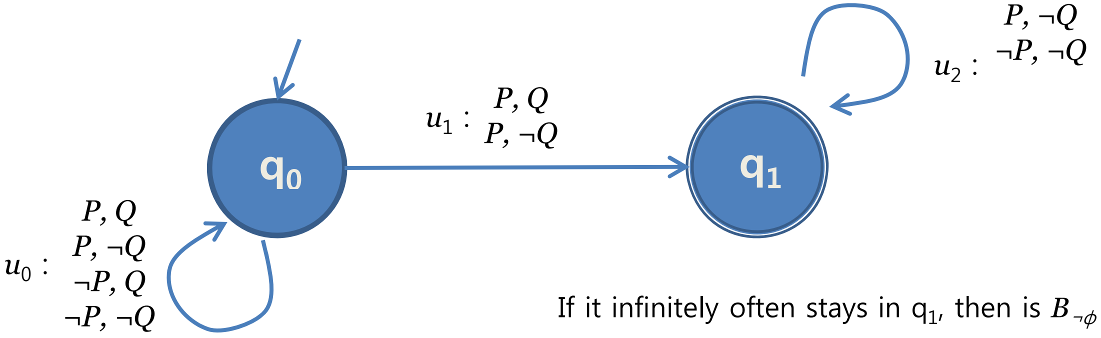
- A simple example
  - ϕ : G($P \Rightarrow$ XF Q) → any occurrence of P must be followed (later) by an occurrence of Q
  - $B_{\neg\phi}$ → there exists an occurrence of P after which we will never again encounter Q


q0
q1
u_1 : P, Q / $P, \neg Q$
u_2 : $P, \neg Q$ / $\neg P, \neg Q$
u_0 : P, Q / $P, \neg Q$ / $\neg P, Q$ / $\neg P, \neg Q$

If it infinitely often stays in q_1, then is $B_{\neg\phi}$ satisfied.

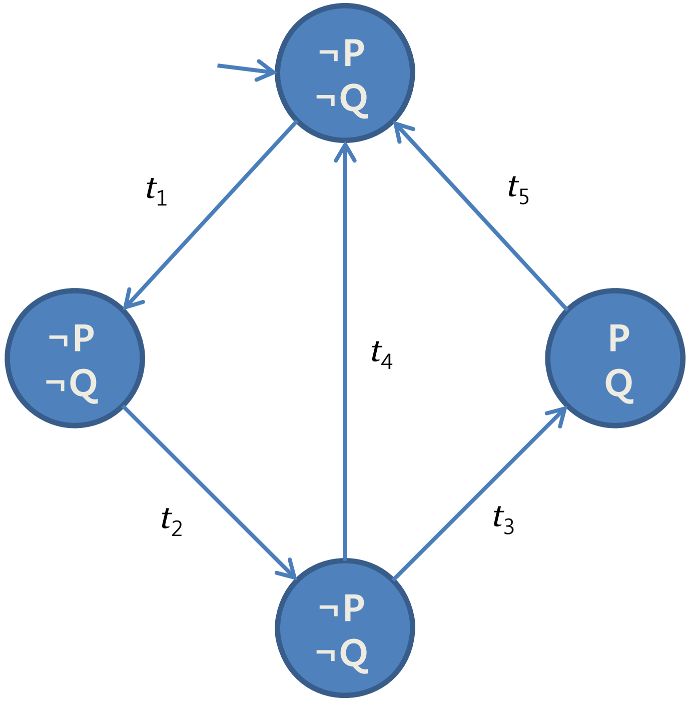
ϕ : G($P \Rightarrow$ XF Q)

$B_{\neg\phi}$:


q₀
q₁
u_0 : P, Q / $P, \neg Q$ / $\neg P, Q$ / $\neg P, \neg Q$
u_1 : P, Q / $P, \neg Q$
u_2 : $P, \neg Q$ / $\neg P, \neg Q$

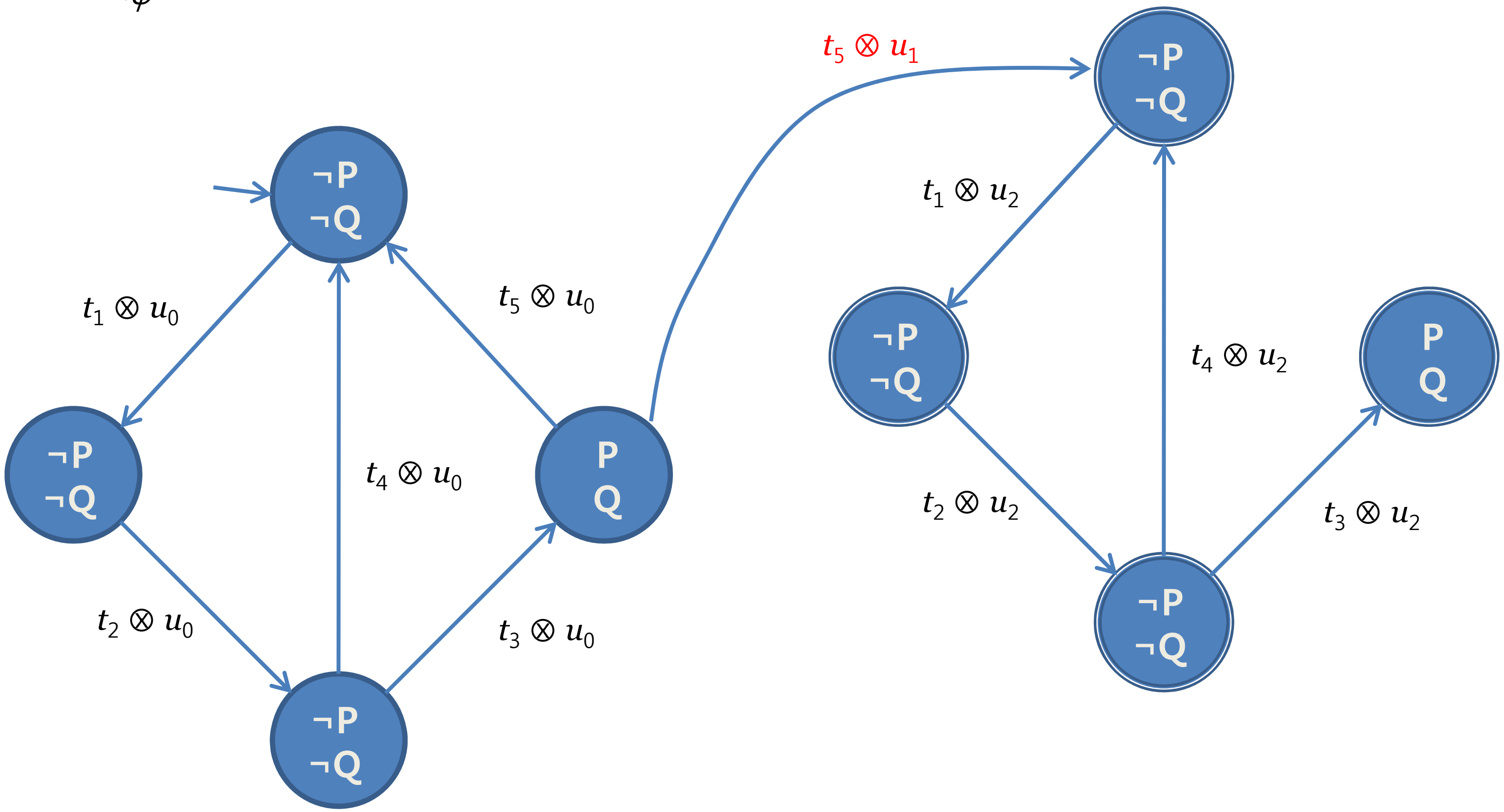
If it infinitely often stays in q$_1$, then is $B_{\neg\phi}$ satisfied.

A :



" does $A \models \phi$? "

$A \otimes B_{\neg\phi}$:



There are behaviors of A accepted by $A \otimes B_{\neg\phi}$

→ The language recognized by $A \otimes B_{\neg\phi}$ is nonempty

→ $A \nvDash \phi$

- Construction of $B_{\neg\phi}$
  - Very difficult technically
  - Automaton $B_{\neg\phi}$ must in general be able to recognize infinite words
    → Büchi automata

- Complexity of the algorithm
  - $B_{\neg\phi}$ has size $O(2^{|\phi|})$ in the worst case
  - $A \otimes B_{\neg\phi}$ has size $O(|A| \times |B_{\neg\phi}|)$
  - If $A \otimes B_{\neg\phi}$ fits in computer memory, we can determine it in time $O(|A| \times |B_{\neg\phi}|)$
  - Model checking "does A, $q_0 \models \phi$?" for a PLTL formula ϕ can be done in time $O(|A| \times 2^{|\phi|})$

- Reachability analysis
  - We can say that $B_{\neg\phi}$ observes the behavior of A when the two automata are synchronized.
  - Observable automata = formal specification of the desired property
    - UPPAAL
    - SPIN

3.3 The State Explosion Problem

- State explosion problem
  - The main obstacle encountered by model checking algorithms
  - Indeed, the algorithms rely on explicit construction of the automaton A
    - Traversal and marking (in case of CTL)
    - Synchronization with $B_{\neg\phi}$ and seeking of reachable states and loops (in case of PLTL)
  - In practice, the number of states of A is quickly very large
  - If we use values that are not priori bounded (integers, a waiting queue, etc.), we cannot even apply it
  - Explicit model checking → Symbolic model checking (Chapter 4)

Chapter 4. Symbolic Model Checking

4. Symbolic Model Checking

- Symbolic model checking
  - Any model checking method attempting to represent symbolically states and transitions
  - A particular symbolic method in which BDDs are used to represent the state variables
    - BDD : Binary Decision Diagram

- Motivation:
  - State explosion is the main problem for CTL or PLTL model checking
  - State explosion occurs whenever we represent explicitly all states of automaton we use

  - Represent very large sets of states concisely, as if they were in bulk.

- Organization of chapter 4
  - Symbolic Computation of State Sets
  - Binary Decision Diagrams (BDD)
  - Representing Automata by BDDs
  - BDD-based Model Checking

4.1 Symbolic Computation of State Sets

- Iterative computation of $Sat(\phi)$
  - $A = <Q, T, \ldots >$
  - $Pre(S)$: immediate predecessors of the states belonging to S in Q
  - $Sat(\phi)$: set of states of A which satisfy ϕ
  - ψ is the sub-formulas of ϕ
  - $Sat(\neg\psi) = Q \setminus Sat(\psi)$
  - $Sat(\psi \wedge \psi') = Sat(\psi) \cap Sat(\psi')$
  - $Sat(EX\ \psi) = Pre(Sat(\psi))$
  - $Sat(AX\ \psi) = Q \setminus Pre(Q \setminus Sat(\psi))$
  - $Sat(EF\ \psi) = Pre^*(Sat(\psi))$
  - … (others are defined in a similar way)

```
/* ==== Computation of Pre*(S) ==== */
X := S;
Y := { };
while (Y != X) {
   Y := X;
   X := X ∨ Pre(X);
}
return X;
```

  - The algorithms in Section 3.1 is an particular implementation of $Sat(\phi)$
  - Hence, $Sat(\phi)$ is an <u>explicit representation</u> of the state sets

- Which symbolic representations to use ?
  - We have to access the following primitives:
    1. A symbolic representation of $Sat(P)$ for each proposition $P \in Prop$,
    2. An algorithm to compute a symbolic representation of $Pre(S)$ from a symbolic representation of S,
    3. Algorithms to compute the complement, the union, and the intersection of the symbolic representations of the sets,
    4. An algorithm to tell whether two symbolic representations represent the same set.

- Which logic for symbolic model checking?
  - Logics based on state formulas
  - CTL is the best.
  - Mu-calculus on tree is possible.

- Systems with infinitely many states
  - Symbolic approach naturally extends to infinite systems.
  - New difficulties:
    1. Much trickier to come up with symbolic representations
    2. Iterative computation $Sat(\phi)$ is no longer guaranteed to terminate.

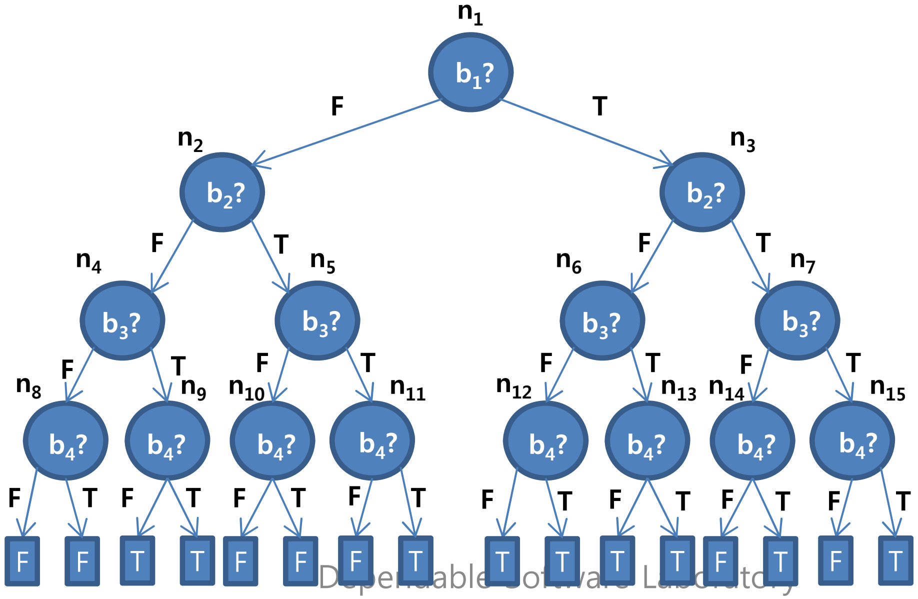
4.2 Binary Decision Diagram (BDD)

- BDD
  - A particular data structure very commonly used for representing states sets symbolically
  - Proposed in 1980s ~ early in 1990s

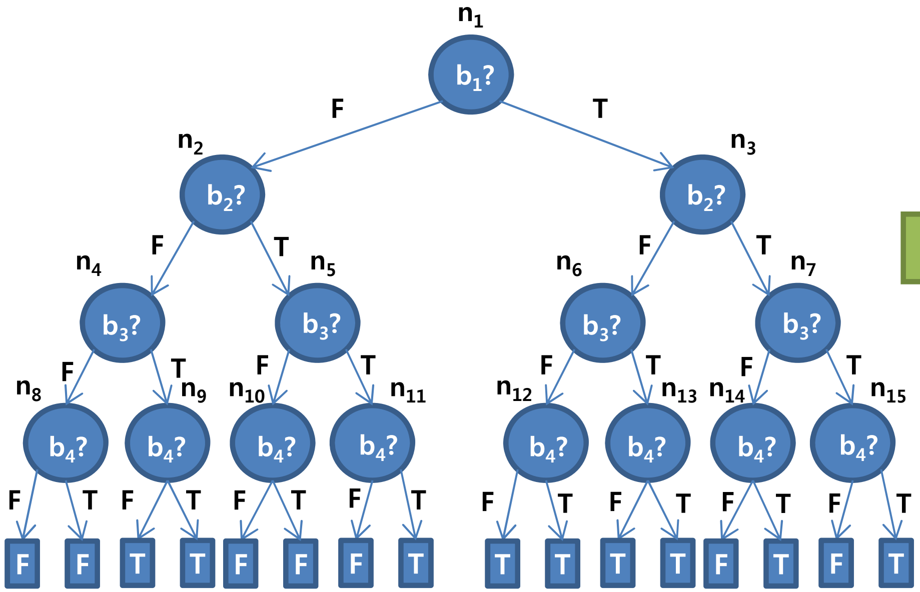
  - Make possible the verification of the system which cannot represent explicitly.

  - Advantages:
    1. Efficiency
    2. Simplicity
    3. Easy Adaptation
    4. Generality

- BDD structure
  - Example
    - Consider n boolean variables $x_1, x_2, \dots , x_n$ associated with a tuple $< b_1, b_2, \dots , b_n >$
    - Suppose $n = 4$,
    - The set S of our interest is the set such that $(b_1 \vee b_3) \wedge (b_2 \Rightarrow b_4)$ is true.
    - We have several ways to represent the set:
      - S = {<F,F,T,F>, <F,F,T,T> , … >
      - S = $(b_1 \vee b_2) \wedge (b_3 \Rightarrow b_4)$
      - S = $(b_1 \wedge \neg b_2) \vee (b_1 \wedge b_4) \vee (b_3 \wedge \neg b_2) \vee (b_3 \wedge b_4)$ ← DNF
      - …
      - Decision Tree ← Our choice.

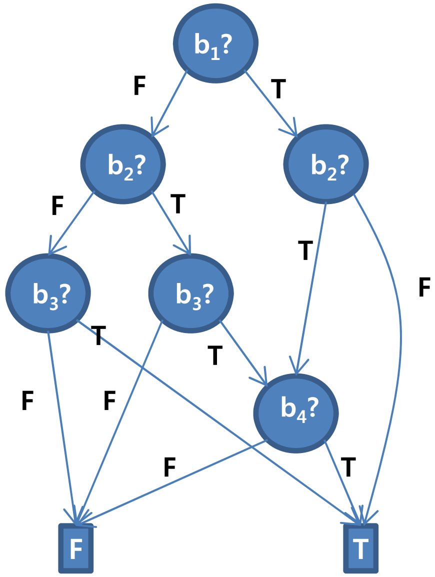


- Decision tree reduction
  - A BDD is a reduced decision tree.
  - Reduction rules:
    1. Identical sub-trees are identified and shared. (n_8 and n_{10})
       → leads to a directed acyclic graph (dag)
    2. Superfluous internal nodes are deleted. (n_7)
  - Advantages:
    1. Space saving
    2. Canonicity



Decision tree

Reduced



BDD

- Canonicity of BDDs
  - BDDs canonically represent sets of boolean tuples. (fundamental property of BDDs)
  - If the order of the variable x_i is fixed, then there exists a unique BDD for each set S.

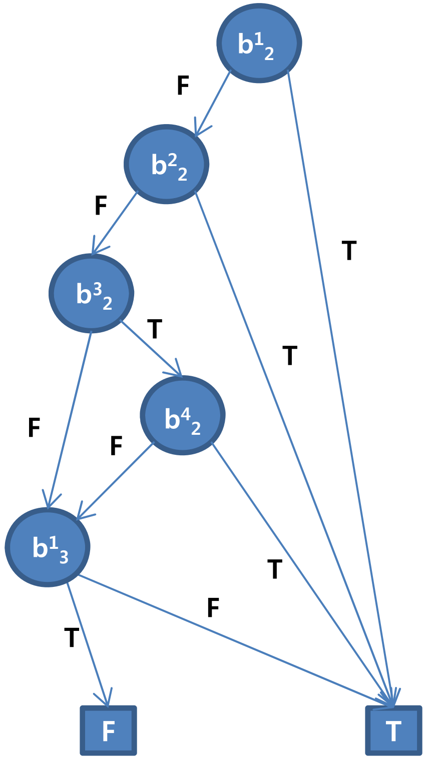
  - Properties of BDDs
    1. We can test the equivalence of two BDDs in constant time.
    2. We can tell whether a BDD represents the empty set simply by verifying whether it is reduced to a unique leaf F.

- Operations on BDDs
  - All boolean operations
    1. Emptiness test
    2. Comparison
    3. Complementation
    4. Intersection
    5. Union and other binary boolean operations
    6. Projection and abstractions
  - Complexity : linear or quadratic (for each operation)
    → the same state explosion problems still exist.

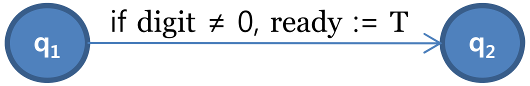
4.3 Representing Automata by BDDs

- Before applying BDDs to symbolic model checking, we need to restate
  - Representing the states by BDDs
  - Representing transitions by BDDs

- Representing the states by BDDs
  - Consider an automaton A with
    - $Q = \{q_0, \ldots, q_6\}$ → b^1_1, b^2_1, b^3_1
    - var digit:0..9 → $b^1_2, b^2_2, b^3_2, b^4_2$
    - var ready:bool → b^1_3
    - $< b^1_1, b^2_1, b^3_1, b^1_2, b^2_2, b^3_2, b^4_2, b^1_3 >$
    - $<$ F, T, T, T, F, F, F, F $> = <q_3$, 8, F $>$
  - Let's represent $Sat(\text{ready} \Rightarrow (\text{digit} > 2))$
    - States $<q, k, b>$ such that if b = T and $k > 2$
    - ready ⇒ (digit > 2) ≡ ¬ ready ∨ (digit > 2)



- Representing transitions by BDDs
  - The same idea is applied.
  - $<q_3, 8, \mathrm{F}> \rightarrow <q_5, 0, \mathrm{F}>$: < F, T, T, T, F, F, F, F, T, F, T, F, F, F, F, F >
  - For example,



  - $(<q, k, b>, <q', k', b'>)$
    → $q = q_1, k \neq 0, q' = q_2, k' = k\ , b' = \mathrm{T}$

    → $(\neg b^1_1 \wedge \neg b^2_1 \wedge b^3_1)$
    $\wedge (b^1_2 \vee b^2_2 \vee b^3_2 \vee b^4_2)$
    $\wedge (\neg b'^1_1 \wedge b'^2_1 \wedge \neg b'^3_1)$
    $\wedge (b'^1_2 \Leftrightarrow b^1_2 \wedge b'^2_2 \Leftrightarrow b^2_2 \wedge b'^3_2 \Leftrightarrow b^3_2 \wedge b'^4_2 \Leftrightarrow b^4_2)$
    $\wedge\ b'^1_3$

4.4 BDD-based Model Checking

- BDDs can serve as an instance of symbolic model checking scheme
  - Provide compact representations for the sets of states in an automaton
  - Support the basic sets of operations
  - Computation of $Pre(S)$ in section 4.1 is very simple

- Implementation
  - SMV (chapter 12)
  - Efficiency of BDDs depends on
    - B_T representing the transition relation T (as containing pairs of states)
    - Choice of ordering for the boolean variables
  - Very easy to explode exponentially

- Perspective
  - Widely used from early 1990s
  - Current work on model checking
    - Aiming at applying BDD technology to solve more verification problems (ex. program equivalence)
    - Aiming at extending the limits inherent to BDD-based model checking
  - Widely used throughout the VLSI design industry

Chapter 5. Timed Automata

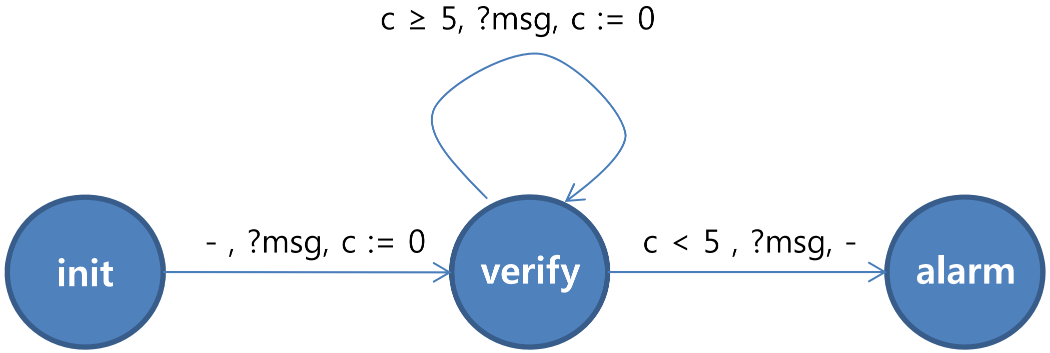
5. Timed Automata

- “Temporal”
  - “Trigger the alarm action upon detecting a problem”
- “Real-Time”
  - “Trigger the alarm less than 5 seconds after detecting a problem”
- Timed Automata
  - Proposed by Alur and Dill in 1994.
  - An answer to this “real-time” needs
- Organization of chapter 5
  - Description of a Timed Automata
  - Networks of Timed Automata and Synchronization
  - Variants and Extensions of the Basic Model
  - Timed Temporal Logic
  - Timed Model Checking

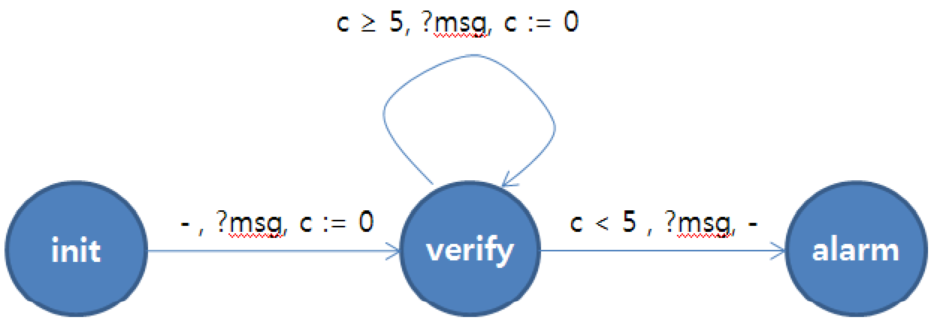
5.1 Description of Timed Automata

- Two fundamental elements of timed automata
  1. A finite automaton (assumed instantaneous between states)
  2. Clocks

- An example



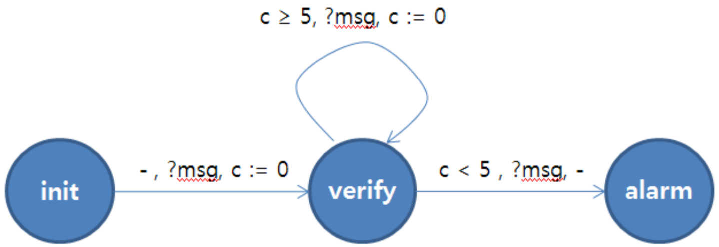
- Clocks and transitions
  - Clocks
    - Variables having non-negative real values in R
    - All clocks are null in the initial system states
    - All clocks evolve at the same speed, synchronously with time
  - Transitions
    - Three items
    - A guard
    - An action (label)
    - Reset of some clocks
  - The system operates as if equipped with
    - A global clock
    - Many individual clocks (each is synchronized with the global clock)



- Configurations and executions
  - Configuration of the system
    - (q, v)
    - q : a current control state of the automaton
    - v : the value of each clock

    - We also refer to v as a valuation of the automaton clocks.
    - Timed automata does not fix the time unit under consideration
  - Execution of the system
    - (usually infinite) sequence of configurations
    - A mapping ρ from R to the set of configuration

    - Configurations change in two ways
      - Delay transition
      - Discrete transition (or action transition)



Discrete transition

$$(\text{init}, 0) \rightarrow (\text{init}, 10.2) \xrightarrow{?msg} (\text{verify}, 0) \rightarrow (\text{verify}, 5.8) \xrightarrow{?msg} (\text{verify}, 0) \rightarrow (\text{verify}, 3.1) \xrightarrow{?msg} (\text{alarm}, 3.1) \rightarrow \ldots$$

Delay transition

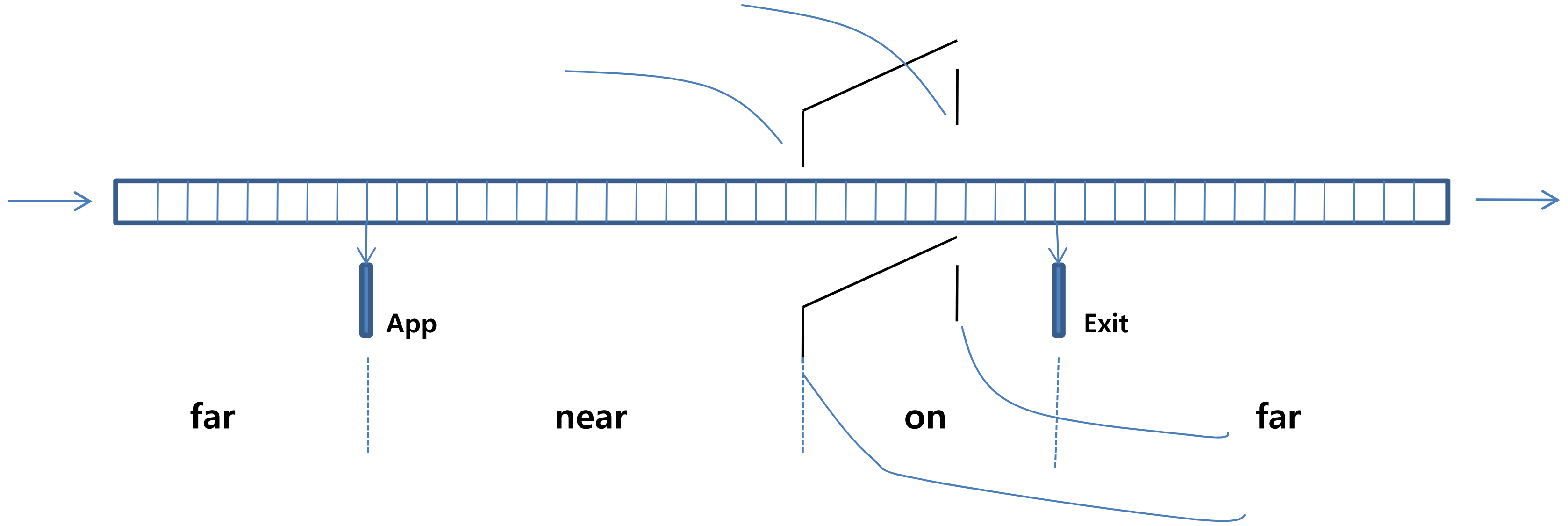
- Trajectory
  - $\rho(0)$: the initial state
  - $\rho(12.3)$ = (verify, 2.1)

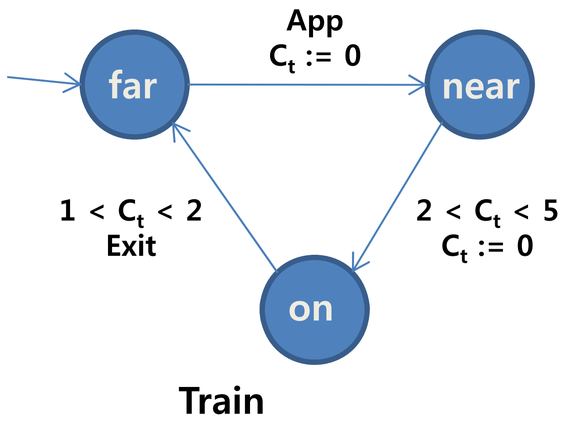
5.2 Networks of Timed Automata and Synchronization

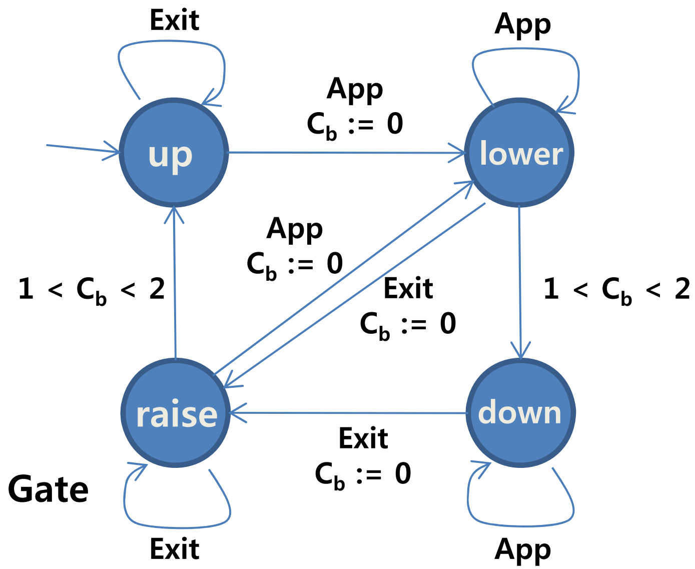
- It is useful to build a timed model in a composite fashion,
  - by combining several parallel automata synchronized with one another → a timed automata network

- Executions of a timed automata network
  - All automata components run in parallel at the same speed
  - Their clocks are all synchronized to the same global clock
  - (q, v) : a network configuration
    - q : a control state vector
    - v : a function associating each network clock with its value at the current time

- Synchronization
  - Timed automata synchronize on transitions (as usually) by resetting the clocks
  - The clocks which were not reset are unchanged
  - No concurrent write conflicts on clocks, since reset writes a zero value and nothing else







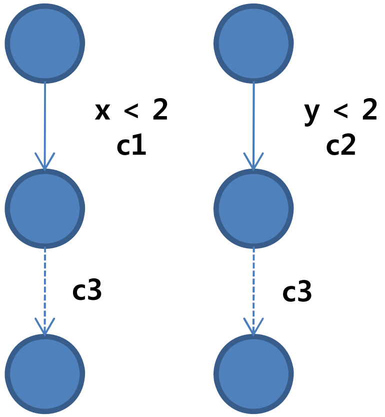
- Example : modeling a railroad crossing

5.3 Variants and Extensions of the Basic Models

- Many variants, and three extensions

1. Invariants
   - Liveness hypothesis in the untimed model
   - Invariant: a state's condition on the clock values, which must always hold in the state
   - Example: near (invariant: $H_t < 5$), on (invariant: $H_t < 2$), lower/raise (invariant: $H_b < 2$)

2. Urgency
   - Used when cannot tolerate a time delay
   - Represented in the system configurations, not in the transitions
   - Allowing urgent/synchronized behaviors in a more natural way



3. Hybrid linear system
   - Models dynamic variables (in a form of differential equations)
   - HyTech

5.4 Timed Temporal Logic

- Given a system described as a network of timed automata,
- We wish to be able to state/verify properties of this system
  - Temporal properties
    - "When the train is inside the crossing, the gate is always closed."
  - Real-time properties
    - "The train always triggers an Exit signal within 7 minutes of having emitted an App signal."

- Three ways to formally state real-time properties
  1. Express it in terms of the reachability of some sets of configurations

  2. Use observer automata in PLTL model checking
     - Given a property ϕ, a network R
     - Testing reachability of some states in the product $R \parallel A_{\phi}$
     - UPPAAL , HYTECH

  3. Use a timed logic
     - TCTL (Timed CTL)
     - Etc.

- TCTL (Timed CTL)

  - Φ , Ψ : : = P_1 | P_2 | … (atomic proposition)
    | $\neg\Phi$ | $\Phi \wedge \Psi$ | $\Phi \Rightarrow \Psi$ | … (boolean combinators)
    | $EF_{(\sim k)}\Phi$ | $EG_{(\sim k)}\Phi$ | $E\Phi\, U_{(\sim k)}\Psi$ (temporal combinators)
    | $AF_{(\sim k)}\Phi$ | $AG_{(\sim k)}\Phi$ | $A\Phi\, U_{(\sim k)}\Psi$ (path quantifiers)

  - ~ : any comparison symbol from {< , ≤ , = , ≥ , >}
  - k : any rational number from Q. (real number)
  - Operator X does not exist in TCTL

  - Example :
    - AG (pb ⇒ $AG_{(\leq 5)}$ alarm)
      - "If a problem occurs, then the alarm will sound immediately and it will sound for at least 5 time units."
    - AG (¬far ⇒ $AF_{(<7)}$ far)
      - "When the train is located in the railway section between the two sensors App and Exit, it will leave this section before 7 time units."

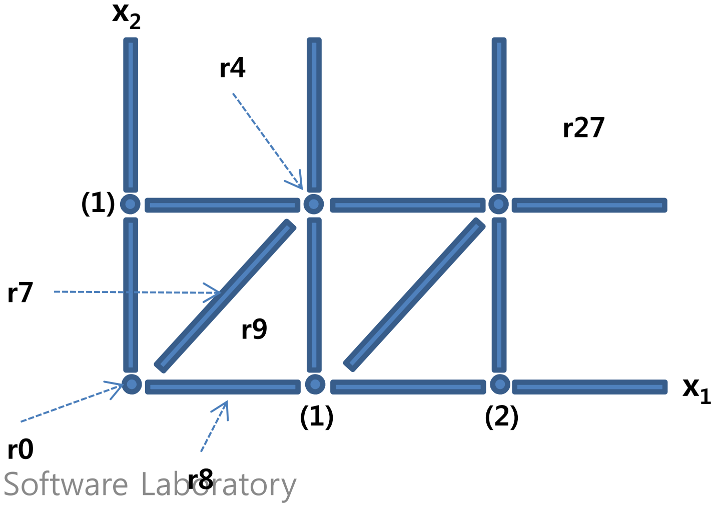
5.5 Timed Model Checking

- With timed automata and TCTL logic
- We wish to obtain a model checking algorithm for them.

- Difficulties : Automaton has an infinite number of configurations, since
  1. Clock values are unbounded
  2. The set of real numbers used in clocks is dense

  → Overcome it with the equivalence classes, called "*regions*"

  – Example: x_1, x_2 ~ k with k = 0, 1, 2
    - 28 regions



- Complexity

  - Model checking algorithms are complicated.
  - The number of regions grows exponentially.

  - $O(n!M^n)$
    - n: number of clocks
    - M: upper bounds of every constant

  - No general and efficient method is likely to exist. (vs. linear complexity in CTL)
  - PSPACE-complete problem

  - Existing tools focus on defining adequate data structures for handing sets of regions → "*zones*"

  - Existing tools have been successfully used
    - HyTech
    - KRONOS
    - UPPAAL

Conclusion of Part I

- Model checking is a verification technique

- It consists of three steps:
  1. Representation of a program or a system by an automaton
  2. Representation of a property by a logical formula
  3. Model checking algorithm

- Model checking is a powerful but restricted tool:
  - Powerfulness: exhaustive and automatic verification
  - Limitation: due to complexity barriers
  - In practice, the size of system is indeed the main obstacle yet to overcome.

- Model checker users are forced to simplify the model under analysis, until it is manageable. (Abstraction)

Part II. Specifying with Temporal Logic

Introduction

- Writing the temporal logic formulas expressing desired system properties

- 4 classification of verification goals
  1. Reachability property
     - Some particular situation *can be reached*.
  2. Safety property
     - Under certain condition, something *never occurs*.
  3. Liveness property
     - Under certain condition, something *will ultimately occur*.
  4. Fairness property
     - Under certain condition, something will (or not) occur *infinitely often*.

  + Deadlock freeness

  + Abstraction methods

Chapter 6. Reachability Properties

Chapter 6. Reachability Properties

- Reachability property
  - Some particular situation can be reached.

  - Examples:
    - (R1) " We can obtain n<0 "
    - (R2) " We can enter a critical section " ← simple
    - (R3) " We cannot have n<0 "
    - (R4) " We cannot reach the crash state " ← negation of the simple
    - (R5) " We can enter the critical section without traversing n=0 " ← with conditional restricts
    - (R6) " We can always return to the initial state " ← stronger / nested
    - (R7) " We can return to the initial state "

- Organization of Chapter 6
  - Reachability in Temporal Logic
  - Model Checkers and Reachability
  - Computation of the Reachability Graph

6.1 Reachability in Temporal Logic

- EF Φ
  - " There exists a path from the current state along which some state satisfying Φ "
  - (R1) " We can obtain n<0 "
    - EF (n<0)
  - (R2) " We can enter a critical section "
    - EF crit_sec
  - (R3) " We cannot have n<0 "
    - ¬EF (n<0)
  - (R4) " We cannot reach the crash state "
    - ¬EF crash
    - AG ¬crash
    - " Along every path, at any time, ¬crash "
  - (R5) " We can enter the critical section without traversing n=0 "
    - E (n≠0) U crit_sec
    - " There exists a path along which n ≠ 0 holds until crit_sec becomes true. "
  - (R6) " We can always return to the initial state "
    - AG (EF init)
  - (R7) " We can return to the initial state "
    - EF init

6.2 Model Checkers and Reachability

- Reachability properties are typically the easiest to verify.
- All model checkers can answer it in principle by simply examining their reachability graph.

- But they do vary in richness.
  - conditional reachability
  - nested reachability
  - etc.

- Design/CPN is specifically designed for reachability property verification.

6.3 Computation of the Reachability Graph

- The effective construction of set of reachable states are non-trivial.
  - Several automata are synchronized.

- Algorithms dealing with reachability problems
  1. Forward chaining
  2. Backward chaining
  3. “On-the-fly” exploration

- Forward chaining
  - A natural approach
  - from initial states → add their successors → until saturation
  - Difficulty: potential explosion of the set constructed

- Backward chaining
  - from target states → add immediate predecessors → until saturation
  - then, test whether some initial states are in there (like $pre^*(S)$ in Section 4.1)
  - Drawback
    1. Target states need to be fixed before.
    2. Computing immediate predecessors is generally more complicated than that of successors.

- "On-the-fly" exploration
  - Explore the reachability graph without actually building it
  - Construction is performed partially, as the exploration proceeds, without remembering everything already visited.
  - Background assumption
    - Present-day computers are more limited in memory resources than in processing speed
  - It is efficient mostly when
    1. Target set is indeed reachable. ("Yes" requires no exhaustive explorations)
    2. Can operate in forward or backward manners (The forward is the traditional)
    3. May apply to some systems with infinitely many states

Chapter 7. Safety Properties

7. Safety Properties

- Safety property
  - Under certain conditions, an (undesirable) event never occur.
  - Examples:
    - (S1) " Both processes will never be in their critical sections simultaneously (mutual exclusion) "
    - (S2) " Memory overflow will never occur "
    - (S3) " The situation … is impossible "
    - (S4) " As long as the key is not in the ignition position, the car won't start " ← with conditions

    - ¬ safety property = reachability property
    - ¬ reachability property = safety property

- Organization of Chapter 7
  - Safety Properties in Temporal Logic
  - A Formal Definition
  - Safety Properties in Practice
  - The history Variables Method

7.1 Safety Properties in Temporal Logic

- AG ¬Φ
  - " Φ never occurs. "
  - (S1) " Both processes will never be in their critical sections simultaneously "
    - AG ¬(crit_sec$_1$ ∧ crit_sec$_2$)
  - (S2) " Memory overflow will never occur "
    - AG ¬overflow
  - (S3) " The situation … is impossible "
    - AG ¬situation
  - (S4) " As long as the key is not in the ignition position, the car won't start "
    - A (¬start W key) (using weak until)
    - A (¬start U key) ← Not a safety property !

7.2 A Formal Definition

- Syntactic characterization
  - Safety properties can be written in the form AG Φ^{-}
    - Φ^{-} is a past temporal formula
  - When a safety property is violated, it should be possible to instantly notice it.
  - We can only notice it, in the current state, relying on events which occurred earlier.

- Temporal logic with past
  - CTL* does not provide past combinators
  - But, we can use a mirror image of future combinators (F^{-1}, X^{-1})

- AG Φ^{-} in practice
  - (S1) AG $\neg(\mathrm{crit\_sec}_1 \wedge \mathrm{crit\_sec}_2)$
    - $\neg(\mathrm{crit\_sec}_1 \wedge \mathrm{crit\_sec}_2)$ is a ϕ^{-}
  - (S4) A $\neg$start W key
    - Can be rewritten in the form: AG (start $\Rightarrow$ F^{-1} key)
    - " It is always true (AG) that if the car starts, then ($\Rightarrow$) the key was inserted beforehand (F^{-1}). "
  - If Ψ_1 and ψ_2 are safety properties, then $\Psi_1 \wedge \psi_2$ again a safety property.
    - But, $\Psi_1 \vee \psi_2$ is in general not

- Safety properties and diagnostic
  - If AG Φ^{-} is not satisfied, then there necessarily exists a finite path leading from *init* to it.
  - Since Φ^{-} is a past formula.

7.3 Safety Properties in Practice

- Safety properties are verified simply by submitting it to a model checker.
- But, in real life, hurdles spring up.

- A simple case: non-reachability
  - The most safety properties
  - $\neg \mathrm{EF}\ (\mathrm{crit\_in}_1 \wedge \mathrm{crit\_in}_2) = \mathrm{AG}\ \Phi^-$
    - $\neg(\mathrm{crit\_in}_1 \wedge \mathrm{crit\_in}_2)$ is a present formula

- Safety without past
  - $\mathrm{A}\ (\neg \mathrm{start}\ \mathrm{W}\ \mathrm{key})$ is used more often than $\mathrm{AG}\ (\mathrm{start} \Rightarrow \mathrm{F}^{-1}\ \mathrm{key})$
  - But, no model checker is able to deal with past formulas. So, mixed logics are used.
  - The problem is their identification.
→ If they are identified, then it can be dealt with similarly
→ Otherwise, we have to use the method of history variables (in section 7.4)

- Safety with explicit past
  - No model checker is able to handle temporal formula with past.
  - Two approaches:
    1. Eliminate the past (in principle, it is possible to translate mixed formulas to pure-future ones)
       - $\mathrm{AG}\ (\phi \Rightarrow \mathrm{F}^{-1}\ \psi) \equiv \mathrm{A}\ (\neg\phi\ \mathrm{W}\ \psi)$, but not easy.
    2. History variable method (section 7.4)

7.4 The History Variables Method

- Skipped !!!

Chapter 8. Liveness Properties

8. Liveness Properties

- Liveness property
  - Under certain conditions, some event will ultimately occur.
  - Some happy event will occur in the end.

  - Examples:
    - (L1) " Any request will ultimately be satisfied "
    - (L2) " By keeping on trying, one will eventually succeed "
    - (L3) " If we call on the elevator, it will bound to arrive eventually "
    - (L4) " The light will turn green (some day regardless of the system behavior)"
    - (L5) " After the rain, the sunshine "
    - (L6) " The program will terminate "

  - Two broad family of liveness properties
    1. Simple liveness : ***progress*** (Chapter 8)
    2. Repeated liveness : ***fairness*** (Chapter 10)

- Organization of Chapter 8
  - Simple Liveness in Temporal Logic
  - Are Liveness Properties Useful?
  - Liveness in the Model, Liveness in the Properties
  - Verification under Liveness Hypotheses
  - Bounded Liveness

8.1 Simple Liveness in Temporal Logic

- F Φ
  - " Φ will ultimately occur. "
  - (L1) " Any request will ultimately be satisfied "
    - AG (req $\Rightarrow$ AF sat)
  - (L7) " The system can always return to its initial state "
    - AG EF init
  - P U Q
    - " Along the execution, we will find a state satisfying Q and P will hold for all the states encountered in the meantime "
    - Regarded as a liveness property
    - $\text{P U Q} \equiv \underbrace{\text{F Q}}_{\text{(liveness)}} \wedge \underbrace{\text{(P W Q)}}_{\text{(safety)}}$
    - A(PUQ) and E(PUQ) are all liveness properties.

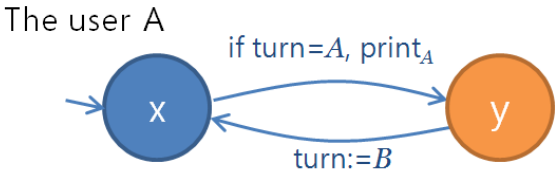
8.2 Are Liveness Properties Useful?

- Abstract liveness properties
  - " If we call on the elevator, it is bound to arrive eventually "
    - It yields no information, from a utilitarian viewpoint.
    - "Abstract" liveness property
  - " An event will occur within at most x time unit "
    - It is useful, but became a safety property.
    - "Bounded" liveness property
  - But, it is still useful
    - "Abstract" more general than "concrete"
    - "Abstract" more efficient than "concrete"
    - "Abstract" and "concrete" are not contradictory

8.3 Liveness in the Model, Liveness in the Properties

- Two different roles in the verification process
  1. Liveness *properties* : we wish to verify
  2. Liveness *hypotheses* : we make on the system model

- When we use a mathematical model(automata) to represent a real system,
  - The semantics of the model in face define *implicit safety and liveness hypotheses*.
  - Safety hypothesis :
    - Clear
    - It can flip from *q* to *q'* only if it includes a transition going from *q* to *q'*.
  - Liveness hypothesis :
    - Not clear
    - The system will chain transitions as long as possible. (to a block state or accepting states)
    - " The system does not terminate without reason, or remain inactive indefinitely without reason. "
    - Can be subtle and cause errors :



In state x, will always end up wishing printing.
→ Different from the real world's behavior !!!

- One must be aware of the premises of the models used and check their adequacy !

8.4 Verification under Liveness Hypotheses

- Verify that specific model behaviors satisfy a given property :
  - ϕ_v : only the model which the liveness hypotheses hold
  - Ψ : a property
  - Verify $\phi_v \Rightarrow \psi$ is sufficient!!!
  - If ψ is a CTL property
    - AF (E PUQ) → A ($\phi_v \Rightarrow$ FE ($\phi_v \wedge$ P U Q))

8.5 Bounded Liveness

- Bounded liveness property
  - A liveness property that comes with a maximal delay which the desired situation must occur.
  - Safety properties from a theoretical viewpoint.
  - Can be rewritten in a form AG ($\psi_2 \Rightarrow F^{-1}\ \psi_1$)
  - Not as important as safety properties

- Bounded liveness in timed systems
  - Often used in the specification of timed systems (in Chapter 5)
  - Explicit constraints on delays → TCTL !!!

  - (BL1) " The program terminates in less than ten seconds "
    - $AF_{<10s}\ \mathrm{end}$ ← bounded liveness property
    - $AG\ (\neg \mathrm{end} \Rightarrow F^{-1}_{<10s}\ \mathrm{start}\ )$ ← safety property

  - (BL2) " Any request is satisfied in less than five minutes "
    - $AG\ (\ \mathrm{req} \Rightarrow AF_{<5m}\ \mathrm{sat}\ )$ ← bounded liveness property
    - $AG\ (\ \neg(F^{-1}_{=5m}\mathrm{req} \wedge G^{-1}_{\leq 5m}\neg \mathrm{sat}\ )$ ← safety property

Chapter 9. Deadlock-freeness

9. Deadlock-freeness

- Deadlock-freeness
  - A special property
  - “ The system can never be in a situation on which no progress is possible. “
  - Correct property relevant for systems that are supposed to run indefinitely.
  - A set of properly identified final states will be required to be deadlock-free.

- Organization of Chapter 9
  - Safety? Liveness?
  - Deadlock-freeness for a Given Automaton
  - Beware of Abstractions!`

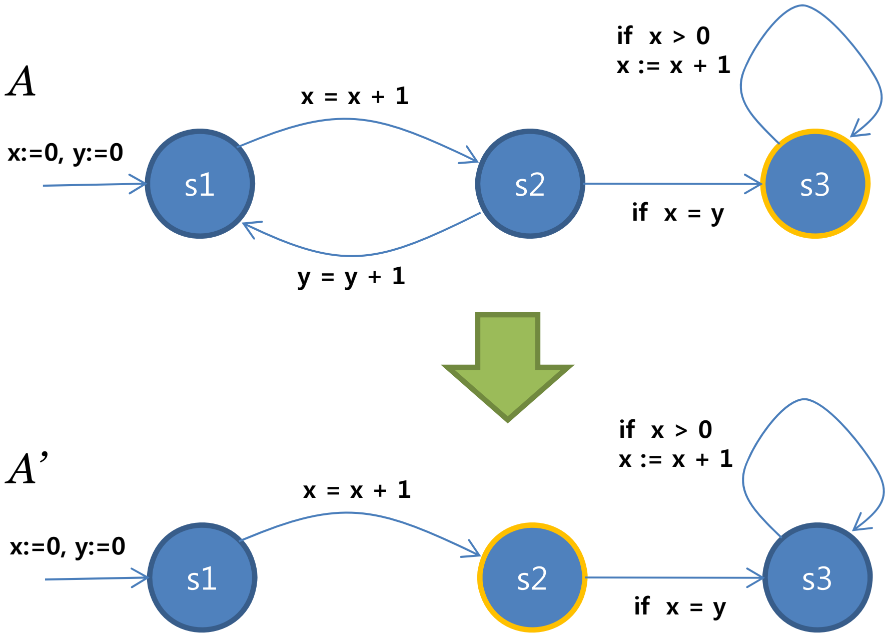
9.1 Safety? Liveness?

- AG EX true
  - " Whatever the state reached may be (AG), there will exist an immediate successor state (EX true) "
  - Not the form of AGϕ^{-1}
  - Deadlock-free is not a safety property.
  - Can be verified if the model checker at our disposal can handle AG EX true.

9.2 Deadlock-freeness for a Given Automaton

- We sometimes think of deadlock-freeness as a safety property
  - For a given automaton, we can describe the deadlock states explicitly.
  - But, it is up to the automaton we obtain.

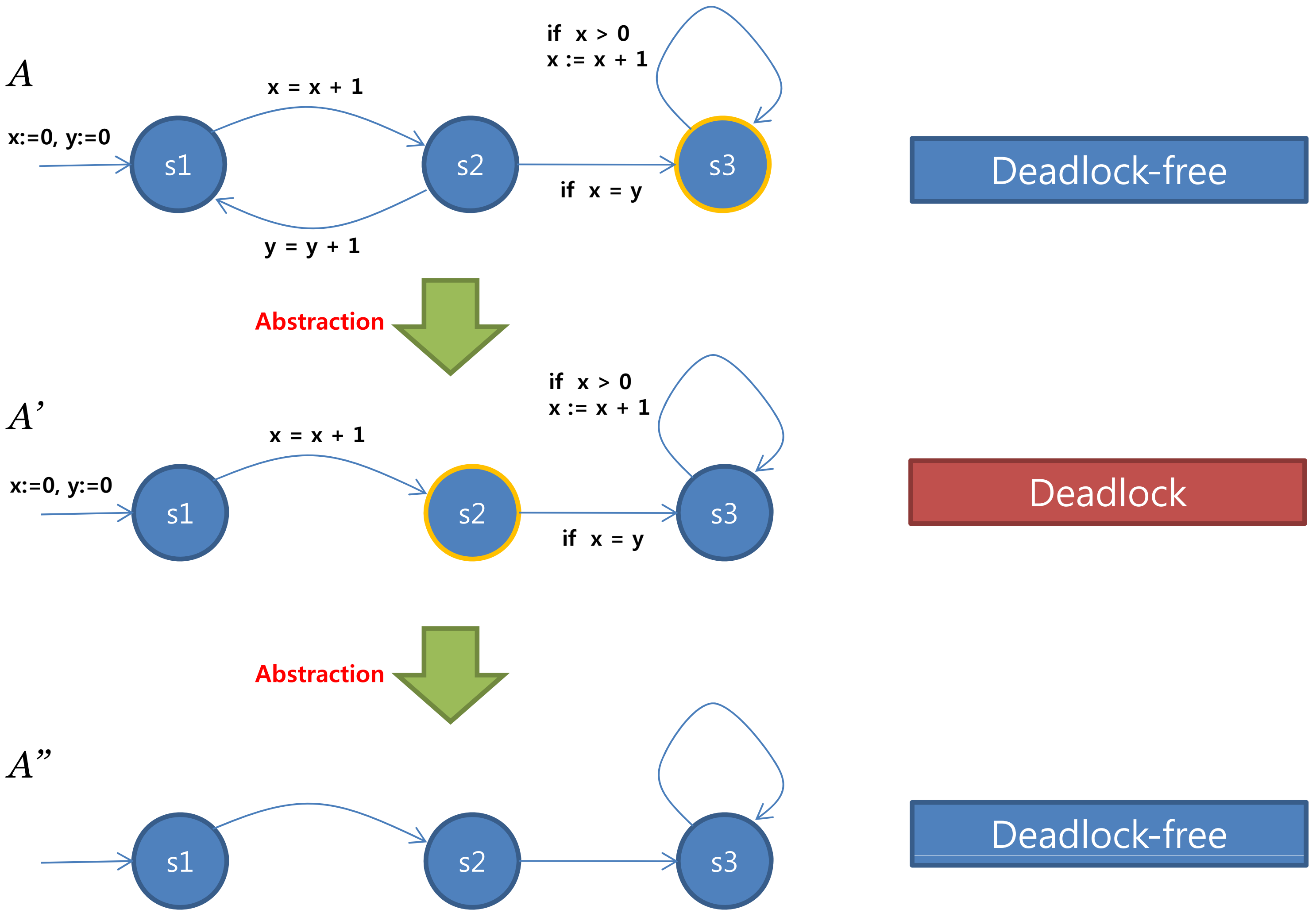
  - For example,



AG EX true → hold! (liveness property)
AG $\neg(s3 \land x \leq 0)$ → hold! (safety property)

AG EX true → not hold! (liveness property)
AG $\neg(s3 \land x \leq 0)$ → hold! (safety property)

9.3 Beware of Abstractions!



Chapter 10. Fairness Properties

10. Fairness Properties

- Fairness Property
  - Under certain conditions, an event will occur (or will fail to occur) infinitely often
  - Examples:
    - (F1) “ The gate will be raised infinitely often”
    - (F2) “ If access to a critical section is infinitely often requested, then access will be granted infinitely often “
  - repeated liveness or repeated reachability

- Organization of Chapter 10
  - Fairness in Temporal Logic
  - Fairness and Nondeterminism
  - Fairness Properties and Fairness Hypothesis
  - Strong Fairness and Weak Fairness
  - Fairness in the Model or in the Property?

10.1 Fairness in Temporal Logic

- GF P
  - " We meet a state in which P holds infinitely often "
  - There is no last state in which P holds.
  - Fairness properties cannot be expressed in pure CTL
    - (F1) " The gate will be raised infinitely often"
      → A (GF $\mathrm{gate\_raised}$)
    - (F2) " If access to a critical section is infinitely often requested, then access will be granted infinitely often "
      → A (GF $\mathrm{crit\_req} \Rightarrow$ FG $\mathrm{crit\_in}$)
  - FCTL or ECTL$^+$
    - CTL + fairness
    - O($|A|$ x $|\phi|^2$)
    - Many tools (like SMV) considers the fairness hypotheses as part of model than choosing FCTL

10.2 Fairness and Nondeterminism

- In practice,
  - Fairness properties are used to describe the form of some nondeterministic sequences
  - " When a nondeterministic choice occurs at some point, it is often assumed to be fair "
  - For example,
    - A die with six faces
    - Its behavior is fair, if it fulfills the property: $A\ (GF\ 1 \wedge GF\ 2 \wedge GF\ 3 \wedge GF\ 4 \wedge GF\ 5 \wedge GF\ 6)$
  - Fairness properties can be viewed as an abstraction of probabilistic properties.

10.3 Fairness Properties and Fairness Hypotheses

- Fairness properties are very often used as hypotheses.

- An example:
  - Classical alternating bit protocol
    - A : a transmitter
    - B : a receiver
    - AB : a line for messages
    - BA : a line for message acknowledgements
    - Messages can be lost → non-deterministic behavior of AB and BA

  - Liveness property : " Any emitted message is eventually received "
    - $G\ (\text{emitted} \Rightarrow F\ \text{received})$
    - Fail !!!
    - The model allows to systematically lose all messages.
    - Our original intension : "unreliable" line, not the whole lose → Fairness hypothesis !!!
    - $A\ (GF\ \neg\text{loss} \Rightarrow G\ (\text{emitted} \Rightarrow F\ \text{received}))$

      fairness hypothesis **liveness property**

  - Repeated liveness property : " If infinitely many messages are emitted, then infinitely many messages will be transmitted "

    repeated liveness property

    - $A\ (GF\ \neg\text{loss} \Rightarrow (GF\ \text{emitted} \Rightarrow GF\ \text{received}))$

      fairness hypothesis **repeated liveness hypothesis**

10.4 Strong Fairness and Weak Fairness

- Fairness property
  - " If P is continually requested, then P will be granted (infinitely often) "

- Weak fairness
  - Assume that P is requested without interruption
  - (FG $request\_P$) ⇒ F P
  - (FG $request\_P$) ⇒ GF P

- Strong fairness
  - Assume that P is requested in an infinitely repeated manner, possibly with interruptions
  - (GF $request\_P$) ⇒ F P
  - (GF $request\_P$) ⇒ GF P

- No difference when using them for model checking of finite systems

10.5 Fairness in the Model or in the Property?

- The best way is
  - Model = automaton + fairness hypotheses
  - Since the second can change independently from the first
  - like SMV model checker

Chapter 11. Abstraction Methods

11. Abstraction Methods

- Abstraction Methods
  - A family of techniques used to simplify automata
  - Simplification aiming at verifying a system (faster) using a model checking approach

  - Examples:
    - (Pb1) “ Does $A \models \phi$? ” ← a complex problem
    - (Pb2) “ Does $A' \models \phi'$? ” ← a much simpler problem

  - “ tricks of the trade “

- Organization of Chapter 11
  - When Is Model Abstraction Required?
  - Abstraction by State Merging
  - What Can Be Proved in the Abstract Automaton?
  - Abstraction on the Variables
  - Abstraction by Restriction
  - Observer Automata

11.1 When Is Model Abstraction Required?

- Two main types of situations for model abstraction
  1. Size of the automaton
     - Too large :
     - Too many variables
     - Too many automata in parallel
     - Too many clocks in the timed automata
  2. Type of the automaton
     - Other types of automata
     - Using integer variables, communication channels, clocks, priorities, etc.

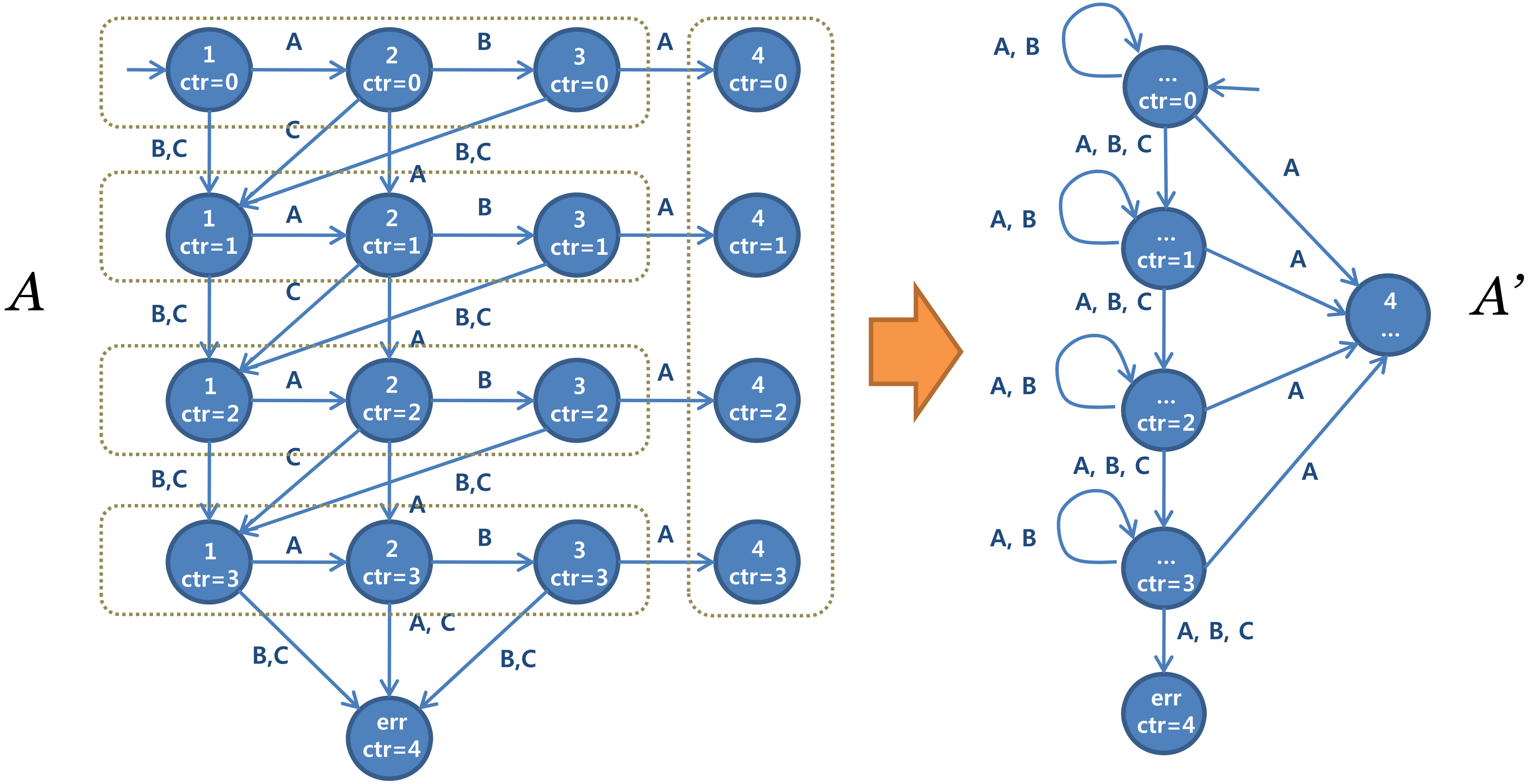
- Three classical abstraction methods
  1. Abstraction by State Merging
  2. Abstraction on the Variables
  3. Abstraction by Restriction

11.2 Abstraction by State Merging

- Folding
  - Viewing some states of an automaton as identical
  - The most important question : Correctness!

  - For example,
    - The digicode door lock with error counters (in Chapter 1)
    - Focusing on the error counter.

    - Correctness problem:
      - All states in A' can be reached through the letter A, but not in A

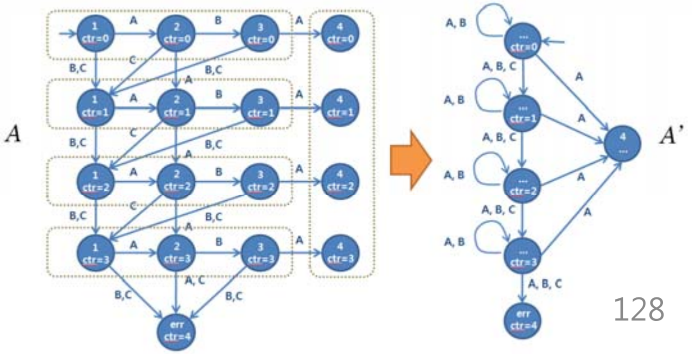

A
1 ctr=0
2 ctr=0
3 ctr=0
4 ctr=0
1 ctr=1
2 ctr=1
3 ctr=1
4 ctr=1
1 ctr=2
2 ctr=2
3 ctr=2
4 ctr=2
1 ctr=3
2 ctr=3
3 ctr=3
4 ctr=3
err ctr=4
A
B
C
B,C
A, C
A'
A, B
A, B, C
… ctr=0
… ctr=1
… ctr=2
… ctr=3
4 …
err ctr=4

11.3 What Can be Proved in the Abstract Automaton?

- We can use state merging to verify safety properties

- Observation (Merging states from A to A')
  1. A' has more behaviors than A.
  2. Now the more behaviors an automaton has, the fewer safety properties it fulfills.
  3. Thus, if A' satisfies a safety property ϕ then a fortiori A satisfies ϕ.
  4. However, if A' does not satisfy ϕ, no conclusions can be drawn about A.

- More behaviors
  - A' has more behaviors than A
  - All executions of A remain present (in folded form) in A'
  - Some new behaviors may be introduced in A'
    - For example, many infinite loops are possible in A'



- Preserving safety properties
  - Necessary to ensure that the property ϕ is indeed a safety property.

- One-way preservation
  - If A' does not satisfies ϕ, then A' satisfies $\neg\phi$.
  - But, in general the negation of a safety property is not a safety property.
  - Abstraction methods are often one-way:
    - If the answer is positive, then is positive too.
    - If the answer is negative, then we learned nothing about A.

- Some necessary precautions
  - Skipped.
  - about the propositions' merging and marking in model checking algorithms

- Modularity
  - State merging is preserved by product.
  - $A' \parallel B$ can be obtained from $A \parallel B$ by a merging operation

- State merging in practice
  - Question : " How will we guess and then specify the sets of states to be merged ? "
  - Answer : " The user is the one who defines and applies his own abstraction. "
    " No tool assistance is offered. "

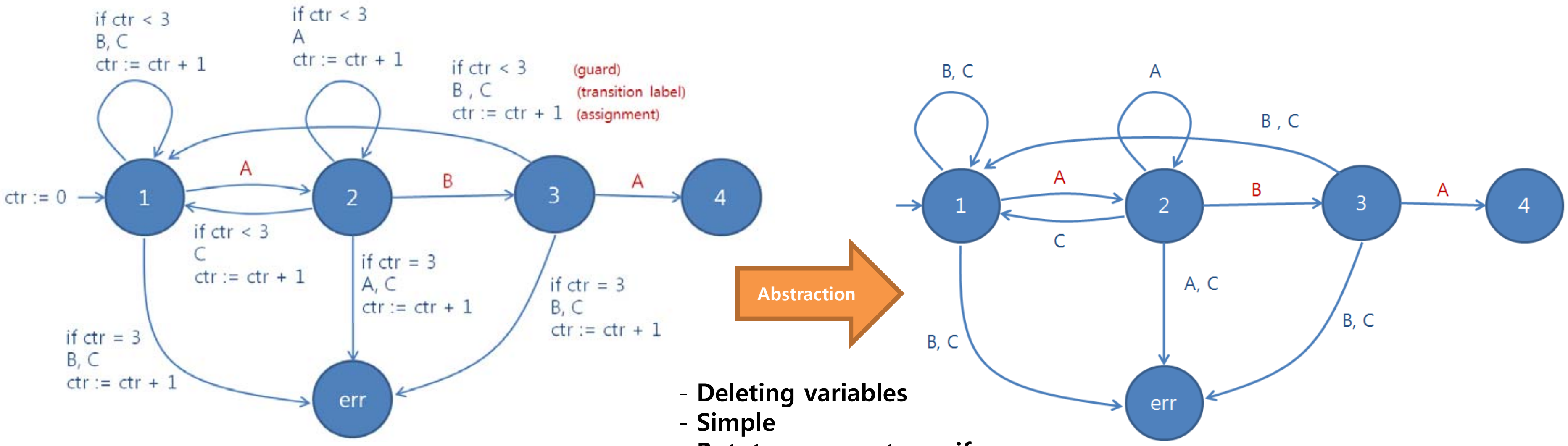
  → Abstraction on variables are often easy to define and implement.

11.4 Abstraction on the Variables

- Abstraction on the variables
  - Concerns the “data” part of automata with variables
  - Directly applies to the description of the automata with variables

- Example

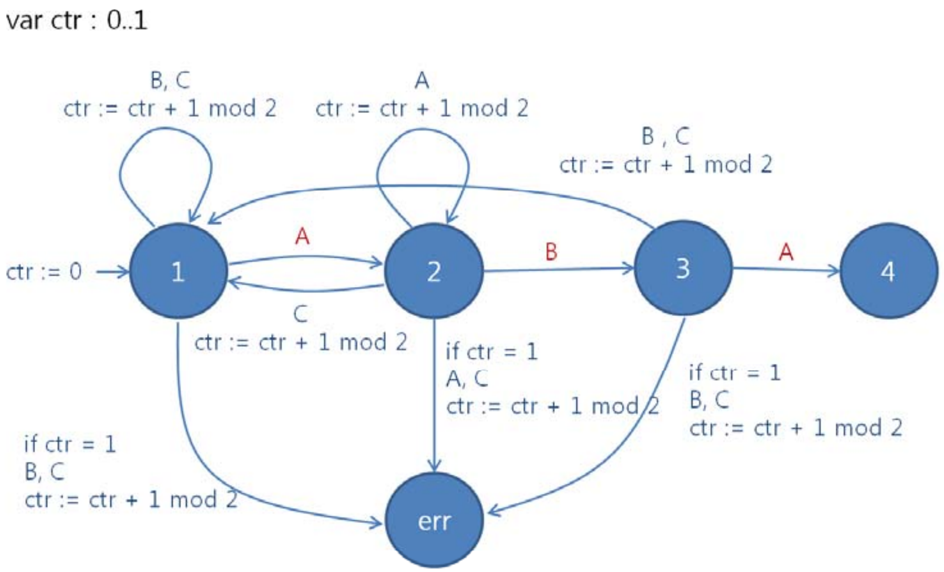
var ctr: int;



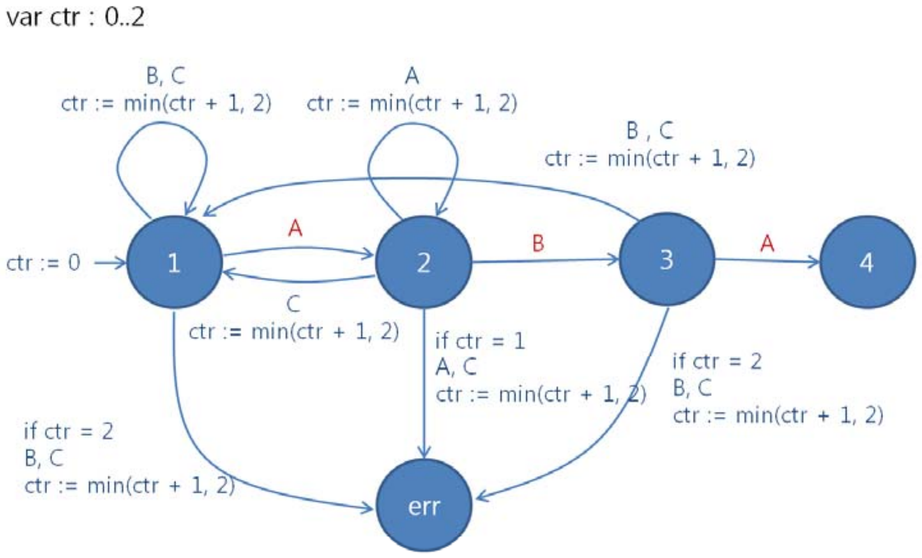
- **Deleting variables**
- **Simple**
- **But, too coarse to verify**

- Abstraction differs from deletion
  - Abstract Interpretation
    - Mathematical theory aiming at defining, analyzing, justifying methods based on abstration

- Bounded variables
  - Narrow down the domain of variables
  - For example,
    - Integer → 0 ~ 10 value
    - The digicode with a modulo 2 counter



The digicode with a modulo 2 counter

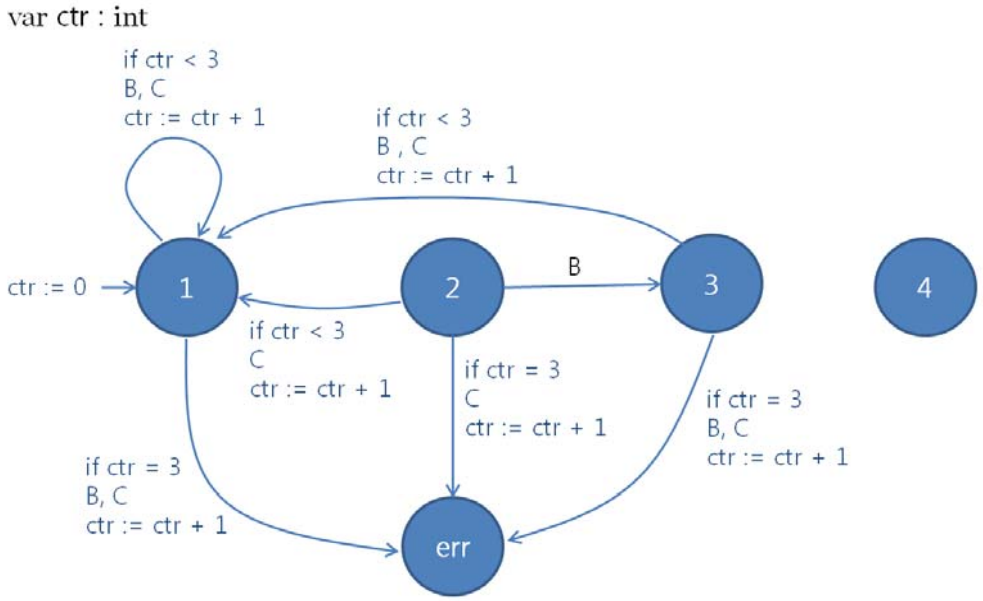


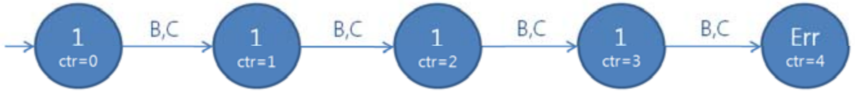
The digicode with a counter bounded by 2

11.5 Abstraction by Restriction

- Restriction
  - A particular form of simplification
  - Operates by forbidding some behaviors of the system or by making some impossible
    - Removing states or transitions
    - Strengthening the guard, etc.

  - For example
    - Remove all the transitions labeled A

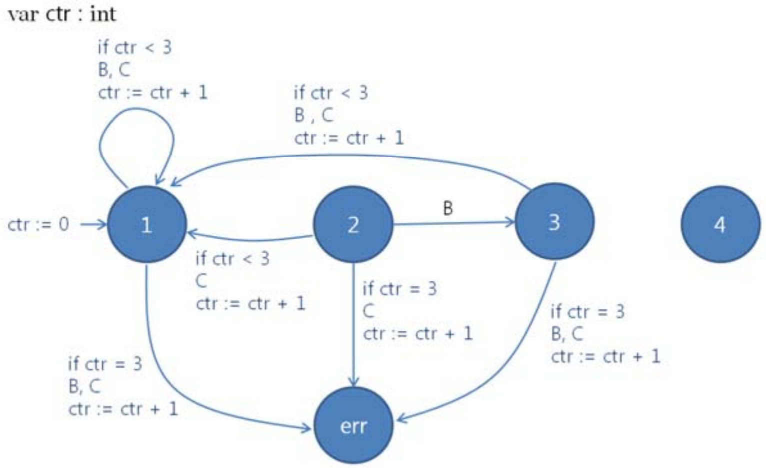




The unfolding of the digicode with no A transition

The digicode with no A transition

- What the restrictions preserve
  - If A' is obtained from A by restriction, then literally all the behaviors of A' are behaviors of A.
  - Thus if A' does not satisfy a safety property, then a fortiori neither does A.
  - Conditional reachability property " EF err " = negation of safety property
  - For example,
    - A' satisfies EF err
    - So we conclude that A also satisfies this property

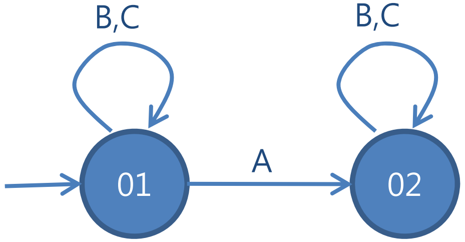


  - Inverse preservation
    - A safety property does not hold.
      (To find errors)
    - But, not to prove the correctness of A
  - Advantage of restriction
    - Simplicity in conceptual and implementational
    - It is a modular operation
    - It naturally applies to an automaton with variables

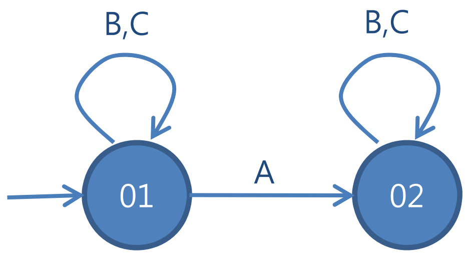
11.6 Observer Automata

- Observer automata
  - Aiming at simplifying a system by restricting its legitimate behaviors to those accepted by an automata outside the system, called observer automata.
  - Reduce the size of automata by restricting its behavior rather than its structure (states and transitions in restriction methods)
  - PLTL model checking algorithm (in Chapter 3) use the concept.

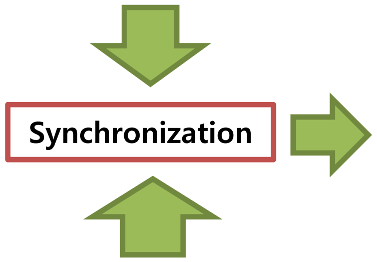
  - An example
    - Supposed that a single A may occur to prove the property.



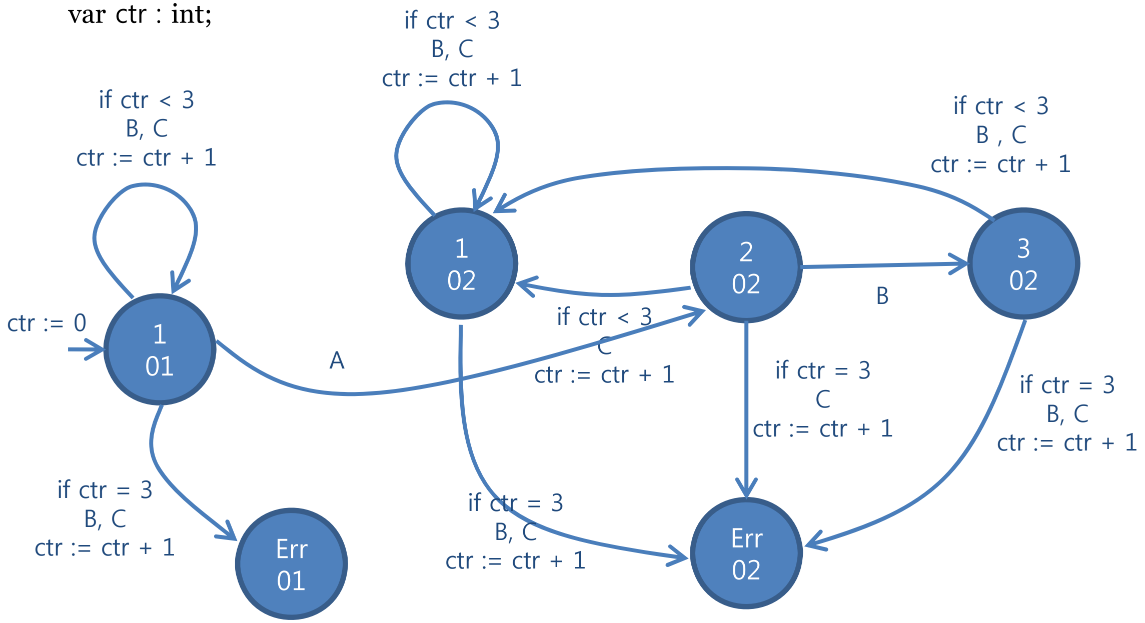
An observer automaton O



An observer automaton O

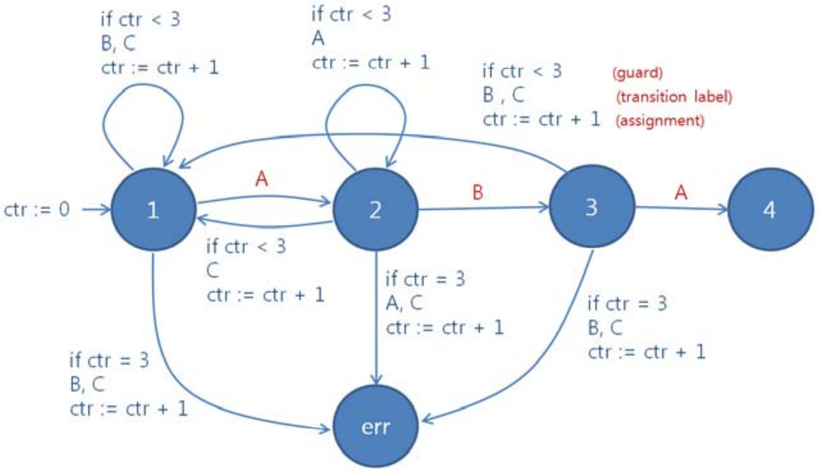


var ctr : int;



The synchronized digicode with its observer

var ctr: int;



An automaton A for the digicode

Part III. Some Tools

Introduction

- 6 tools, concerned with a particular application domain
  - SMV
  - SPIN
  - DESIGN/CPN
  - UPPAAL
  - KRONOS
  - HYTECH

Chapter 12. SMV – Symbolic Model Checking

Chapter 13. SPIN – Communicating Automata

Chapter 14. DESIGN/CPN – Colored Petri Nets

Chapter 15. UPPAAL – Timed Automata

Chapter 16. KRONOS – Model Checking of Real-time Systems

Chapter 17. HYTECH – Linear Hybrid Systems

“Formal Modeling and Verification of Safety-Critical Software implemented in PLC”
- IEEE Software, May/June, 2009.

정형 요구사항 명세 기반 원자력 소프트웨어 개발 방법론

Dependable Software Laboratory

KONKUK University, Korea

http://dslab.konkuk.ac.kr

2010.05.25

정형 요구사항 명세 기반
원자력 소프트웨어 개발 방법론

Dependable Software Laboratory

KONKUK University, Korea

http://dslab.konkuk.ac.kr

SMV를 이용한
NuSCR 정형명세에 대한 정형검증

Dependable Software Laboratory
Konkuk University
http://dslab.konkuk.ac.kr

2010.05.25

SMV Verification for NuSCR
- Demo -

Dependable Software Laboratory
Konkuk University
http://dslab.konkuk.ac.kr

2010.05.25

SPIN을 이용한
MOST NS 프로토콜 정형검증

이동아 학생의 연구내용 삽입 요망

SPIN을 이용한 차량용 MOST Network Service 프로토콜 스택 정형검증

Formal Verification of Protocol Stack for MOST Network Service using SPIN

이동아, 윤상현, 이무열, 진현욱, 유준범

UPPAAL을 이용한 커피자판기 정형명세 및 정형검증

2009 건국대학교 대학원

고급소프트웨어공학 수업

팀프로젝트 T2 / T5

UPPAAL을 이용한 커피자판기 설계

Team 2(이근수, 김준영)

Model 2 검증
(Coffee Vending Machine)

by. 꿀꿀자동차