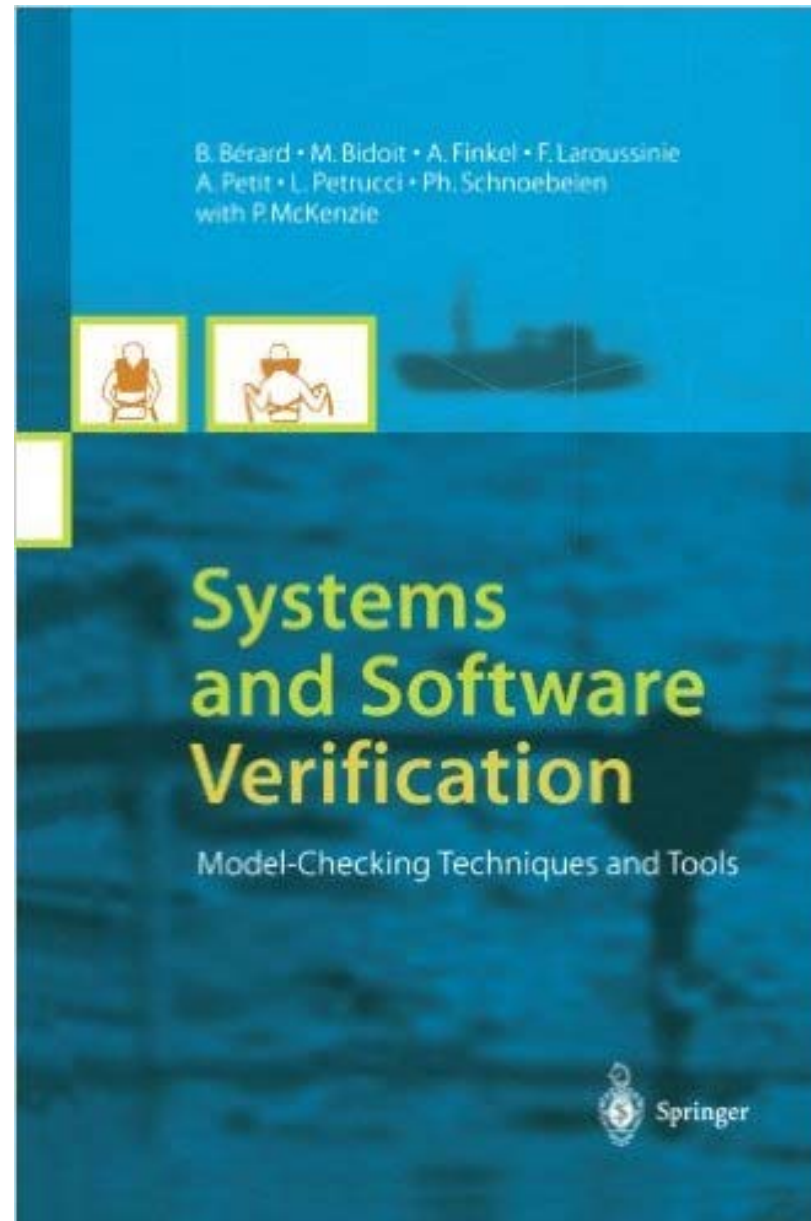


Advanced Software Engineering

Introduction to Model Checking

JUNBEOM YOO
KONKUK University

Text



- Automata
- Temporal Logic
- Model Checking
- Property Patterns

FORMAL VERIFICATION : BASIC

1. Automata

1. Automata

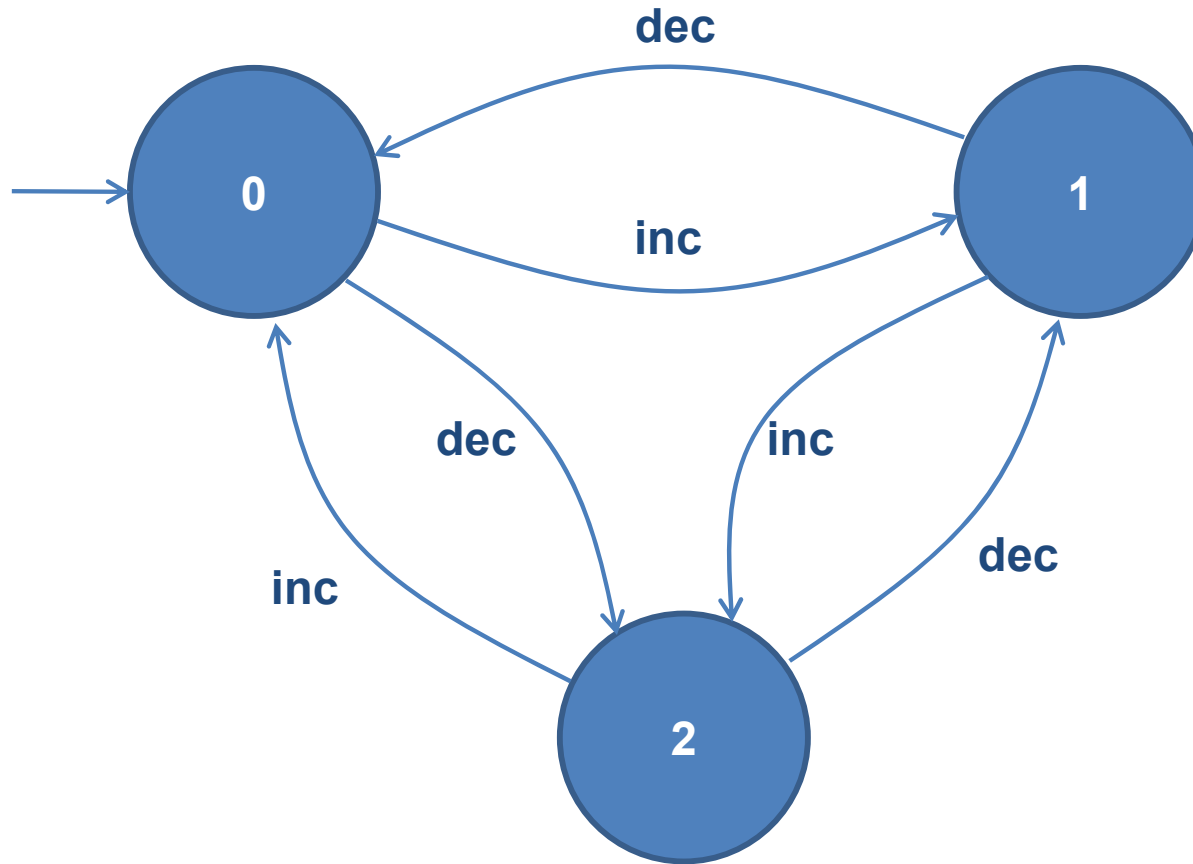
- **Model checking** consists in verifying some properties of the model of a system.
 - Modeling of a system is difficult.
 - No universal method exists to model a system.
 - Best performed by qualified engineers
- This chapter describes a general model which serves as a basis.
- Organization
 - Introductory 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**
 - A machine evolving from one state to another under the action of transitions
 - Graphical representation
 - Best suited for verification by model checking techniques

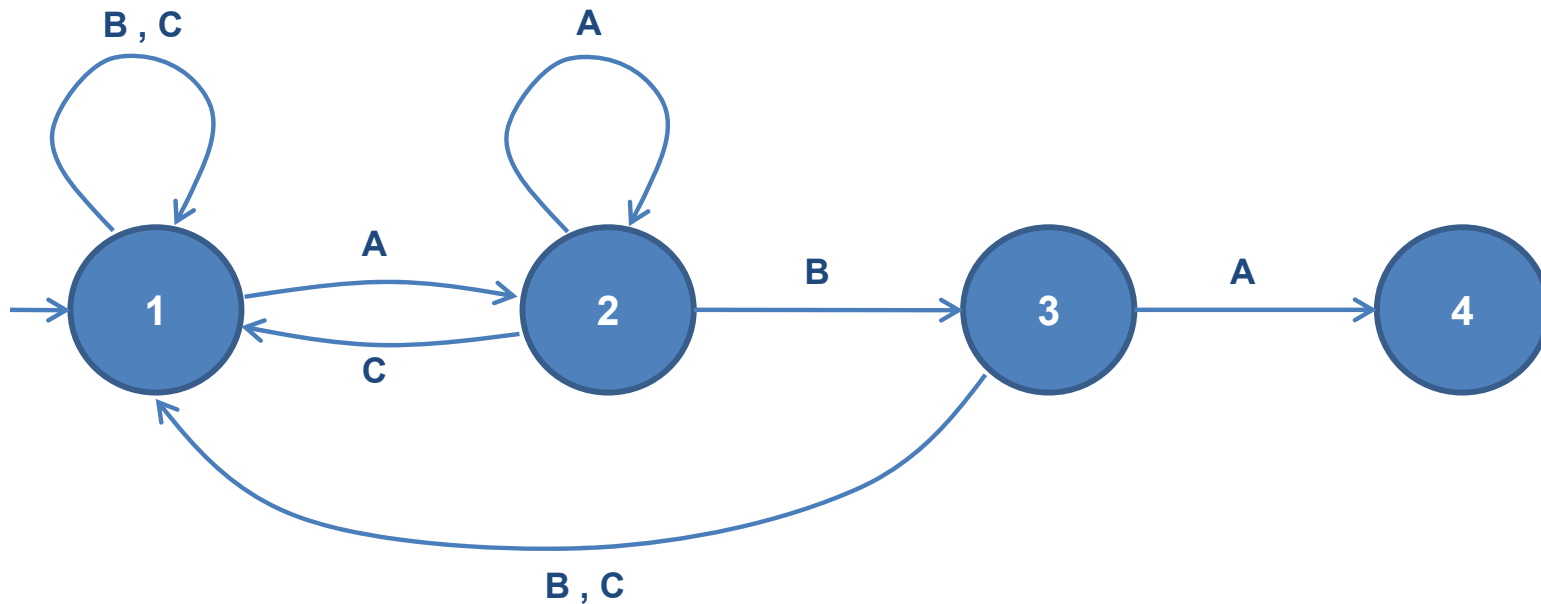


An automate model of a digital watch (24x60=1440 states)



A_{c3} : a modulo 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.
 - Assumptions:
 - 3 keys A, B, and C
 - Correct key sequence : ABA



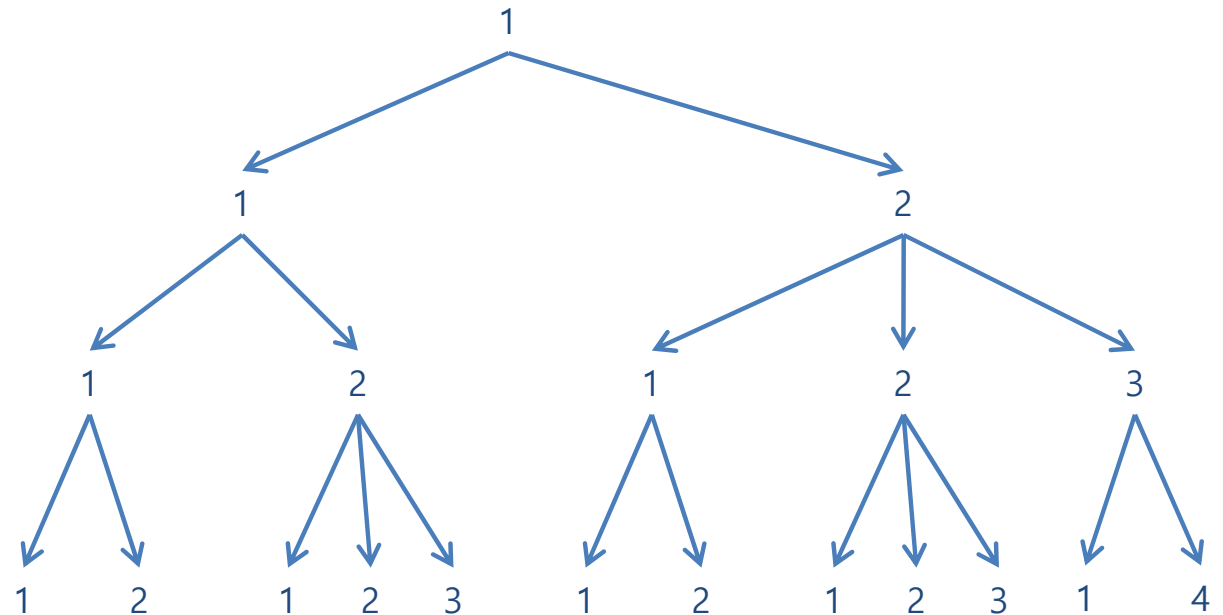
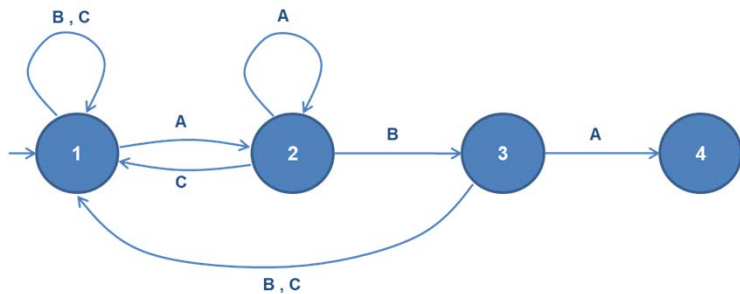
- Two fundamental notations

- **execution**

- A sequence of states describing one possible evolution of the system
 - Ex. 1121 , 12234 , 112312234 ← 3 different executions

- **execution tree**

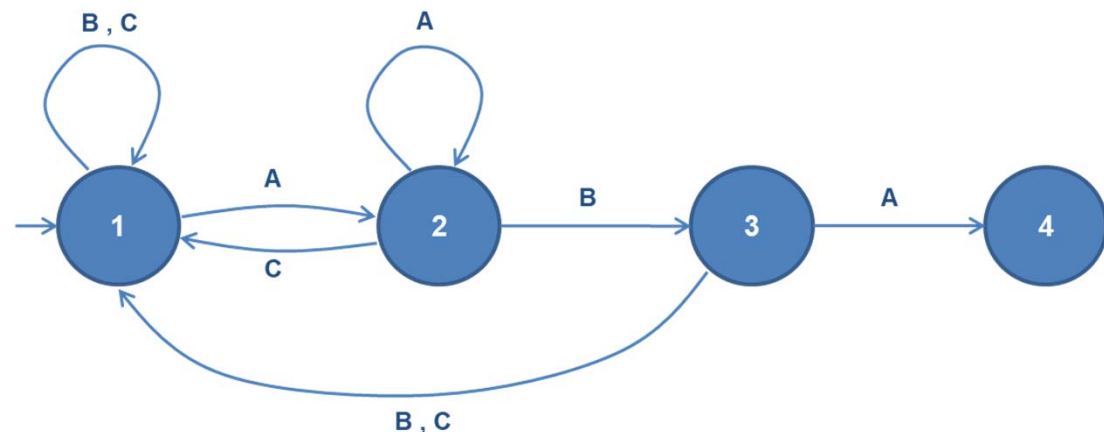
- A set of all possible executions of the system in the form of a tree
 - Ex. 1
 - 11, 12
 - 111, 112, 121, 122, 123
 - 1111, 1112, 1121, 1122, 1123, 1211, 1212, 1221, 1222, 1223, 1231, 1234
 - ...



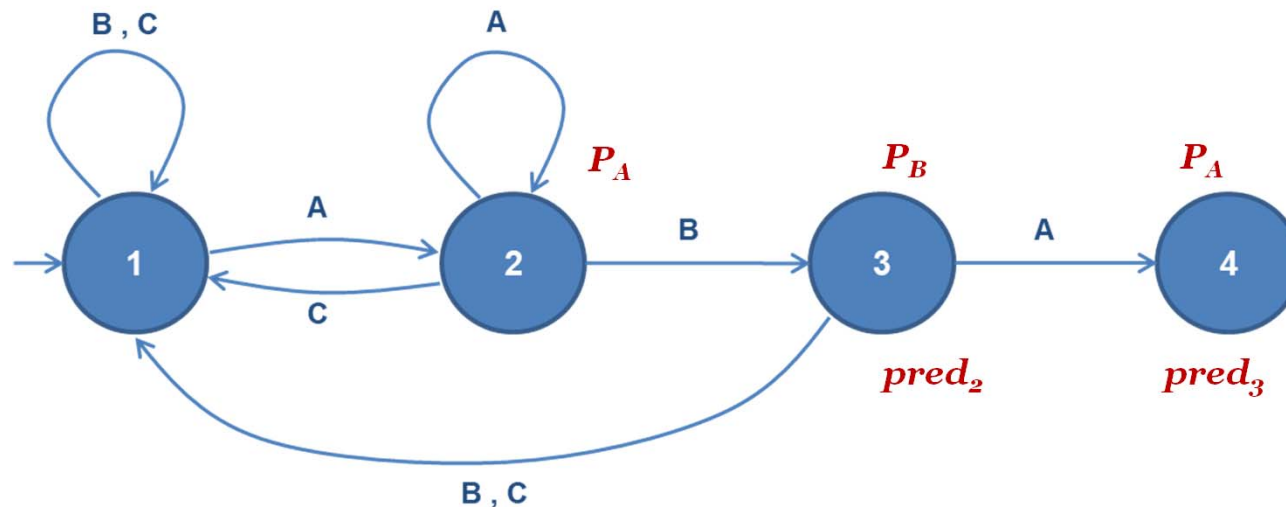
- We now **associate** each automaton state with a number of elementary properties which we know they are satisfies.
 - Since our goal is to verify system model properties.

- **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
 - If the door opens, then A, B, A were the last three letters keyed in, in that order.
 - Keying in any sequence of letters ending in ABA opens the door.
- Let's prove the properties with the propositions



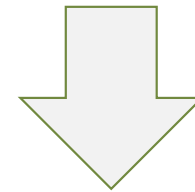
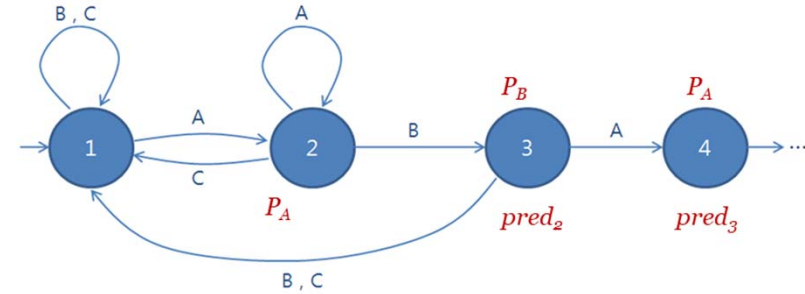
1.2 A Few Definition

- An automaton is a tuple $A = \langle Q, E, T, q_o, l \rangle$ in which
 - Q : a finite set of states
 - E : the finite set of transition labels
 - $T \subseteq Q \times E \times Q$: 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

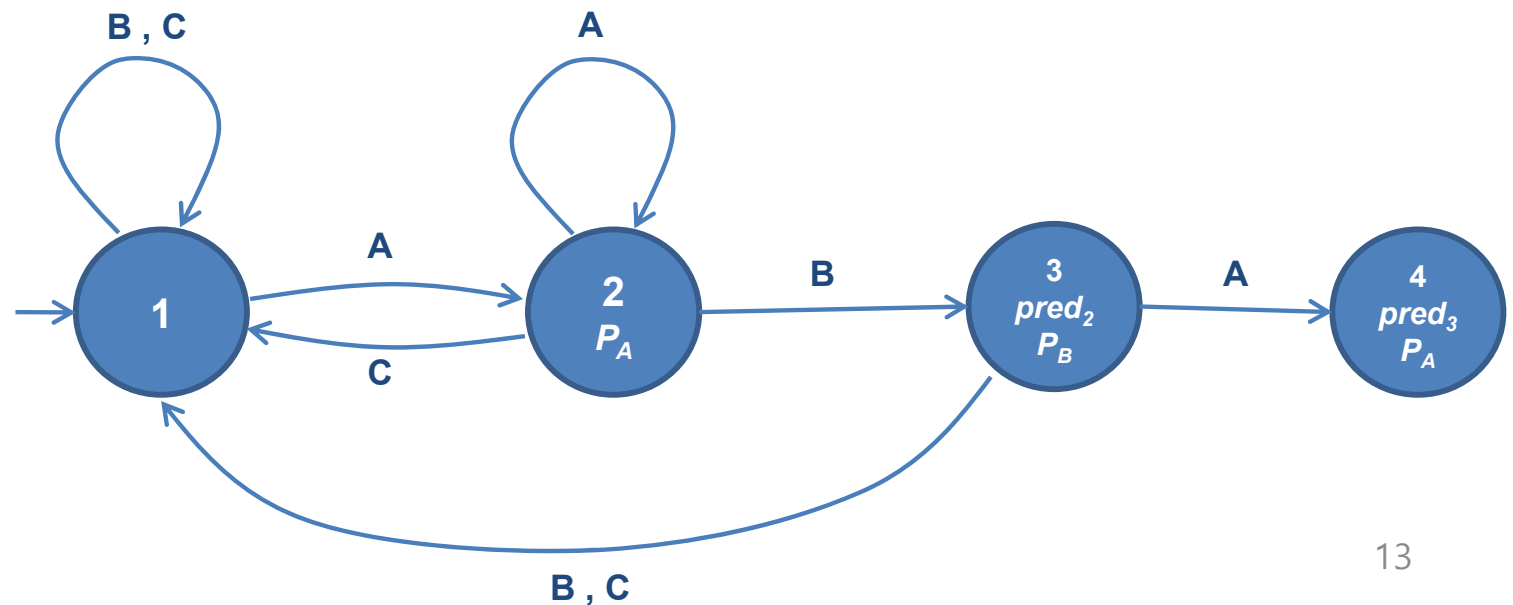
$$A = \langle Q, E, T, q_o, l \rangle$$

- $Q = \{1, 2, 3, 4\}$
- $E = \{A, B, C\}$
- $T = \{ (1,A,2), (1,B,1), (1,C,1), (2,A,2), (2,B,3), (2,C,1), (3,A,4), (3,B,1), (3,C,1) \}$
- $q_o = 1$

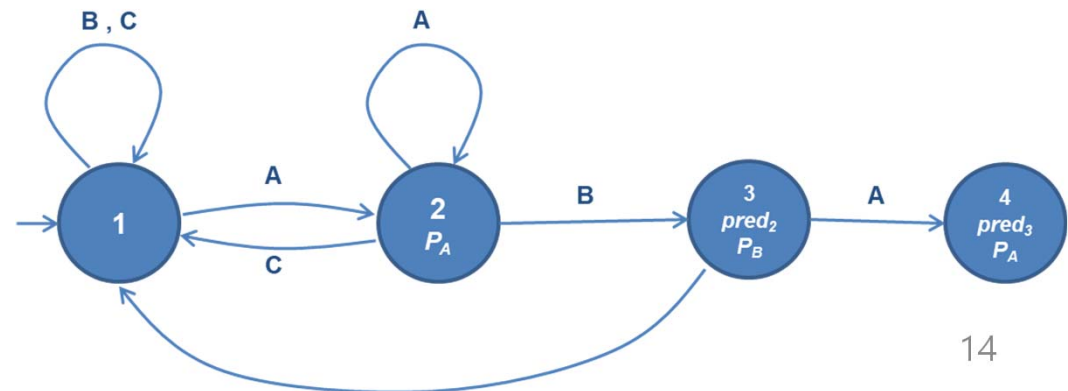
$$l = \begin{cases} 1 \rightarrow \emptyset \\ 2 \rightarrow \{P_A\} \\ 3 \rightarrow \{P_B, pred_2\} \\ 4 \rightarrow \{P_A, pred_3\} \end{cases}$$



The digicode with its atomic propositions



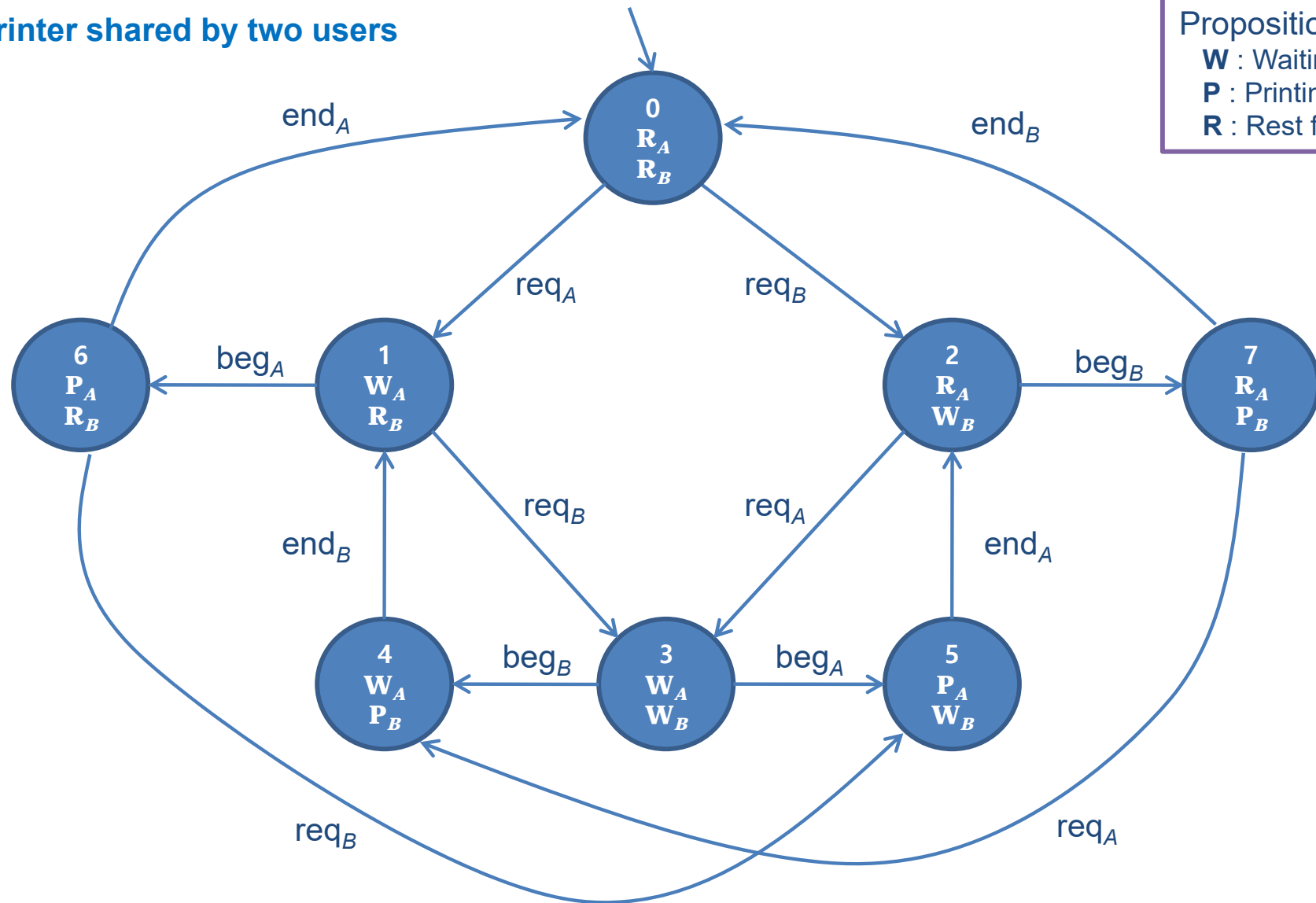
- Formal definitions of automaton's behavior
 - a **path** of automaton A :
 - A sequence σ , finite or infinite, of transitions which follows each other
 - Ex.3 $B \rightarrow 1 \xrightarrow{A} 2 \xrightarrow{A} 2$
 - a **length** of a path σ :
 - $|\sigma|$
 - σ 's potentially infinite number of transitions: $|\sigma| \in N \cup \{\omega\}$
 - a **partial execution** of A :
 - A path starting from the initial state q_0
 - Ex.1 $A \rightarrow 2 \xrightarrow{A} 2 \xrightarrow{B} 3$
 - 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
 - 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



$A = \langle Q, E, T, q_0, l \rangle$

- $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$

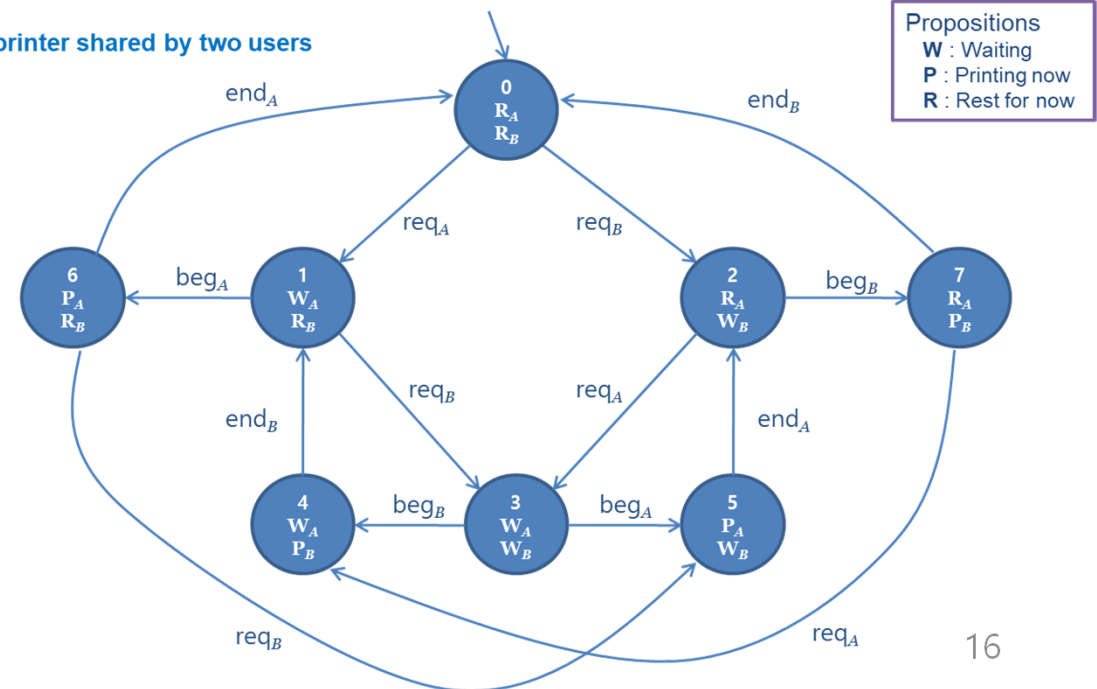
- $E = \{\text{req}_A, \text{req}_B, \text{beg}_A, \text{beg}_B, \text{end}_A, \text{end}_B\}$

- $T = \{ (0, \text{req}_A, 1), (0, \text{req}_B, 2), (1, \text{req}_B, 3), (1, \text{beg}_A, 6), (2, \text{req}_A, 3), (2, \text{beg}_B, 7), (3, \text{beg}_A, 5), (3, \text{beg}_B, 4), (4, \text{end}_B, 1), (5, \text{end}_A, 2), (6, \text{end}_A, 0), (6, \text{req}_B, 5), (7, \text{end}_B, 0), (7, \text{req}_A, 4) \}$

- $q_0 = 0$

- $l = \left\{ \begin{array}{ll} 0 \rightarrow \{R_A, R_B\}, & 1 \rightarrow \{W_A, R_B\} \\ 2 \rightarrow \{R_A, W_B\}, & 3 \rightarrow \{W_A, W_B\} \\ 4 \rightarrow \{W_A, P_B\}, & 5 \rightarrow \{P_A, W_B\} \\ 6 \rightarrow \{P_A, R_B\}, & 7 \rightarrow \{R_A, P_B\} \end{array} \right.$

A printer shared by two users



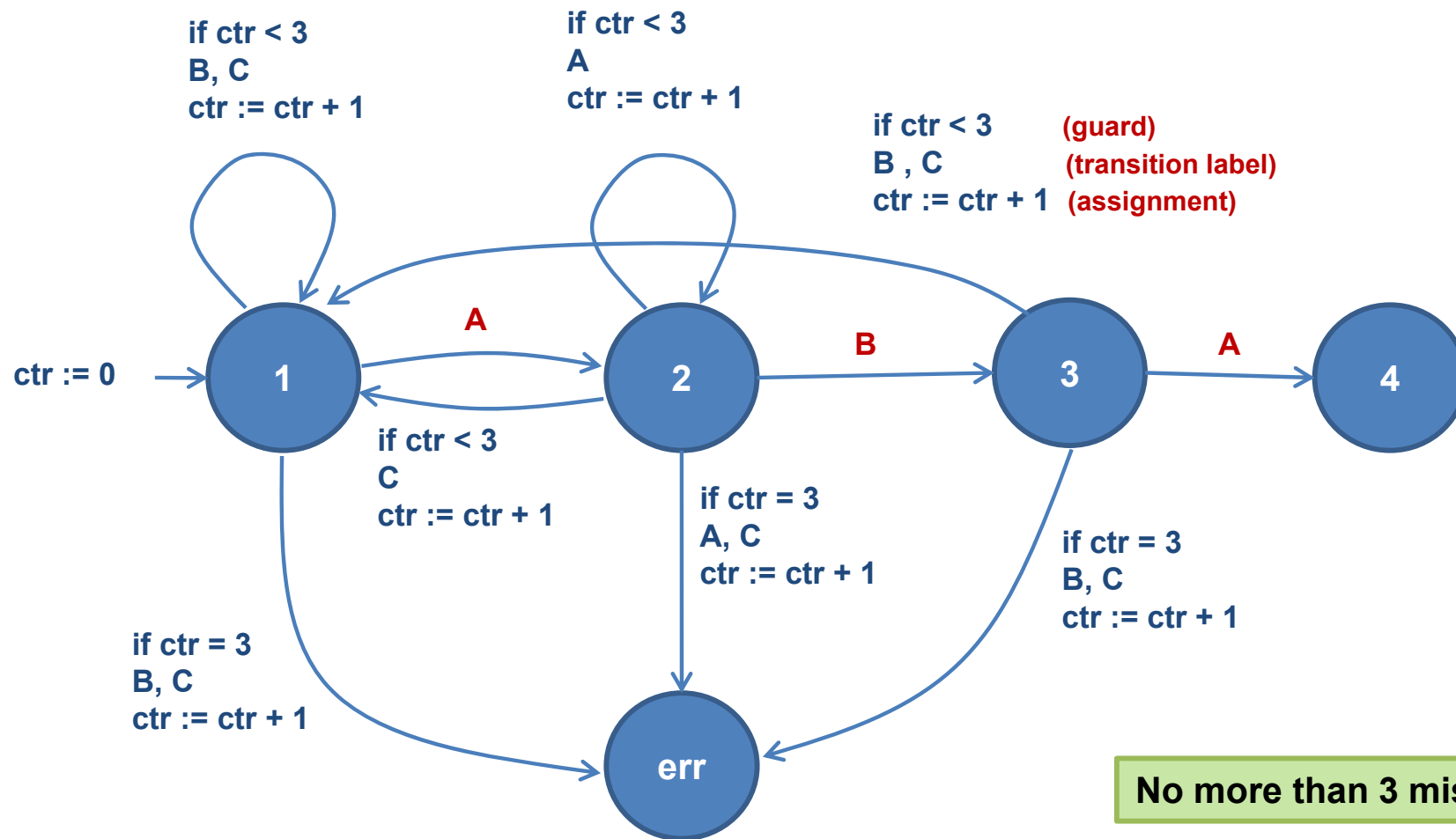
- **Properties** of the printer manager to verify
 1. We would undoubtedly wish to prove that any printing operation is preceded by a print request.
 - In any execution, any state in which P_A holds is preceded by a state in which the proposition W_A holds.
 2. Similarly, we would like to check that any print request is ultimately satisfied. (\rightarrow 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
 - Property 1 is TRUE.
 - Property 2 is FALSE,
 - A counterexample : 0 1 3 4 1 3 4 1 3 4 1 3 4 1 ...

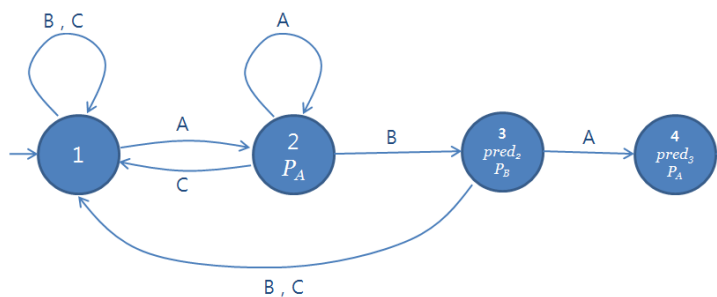
1.4 Few More Variables

- 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 (guarding conditions)

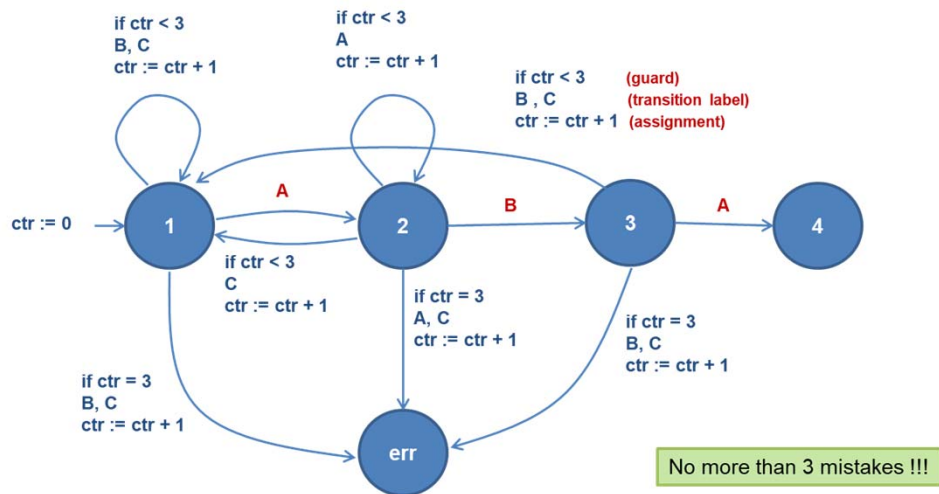


The digicode with guarded transitions



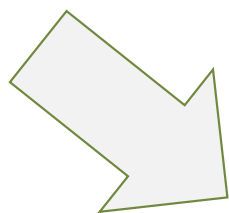
- It is often necessary, in order to apply model checking methods,
 - to **unfold** the behaviors of an automaton with variables
 - into a **state graph**
 - in which the possible transitions appear, and the configurations are clear marked.

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

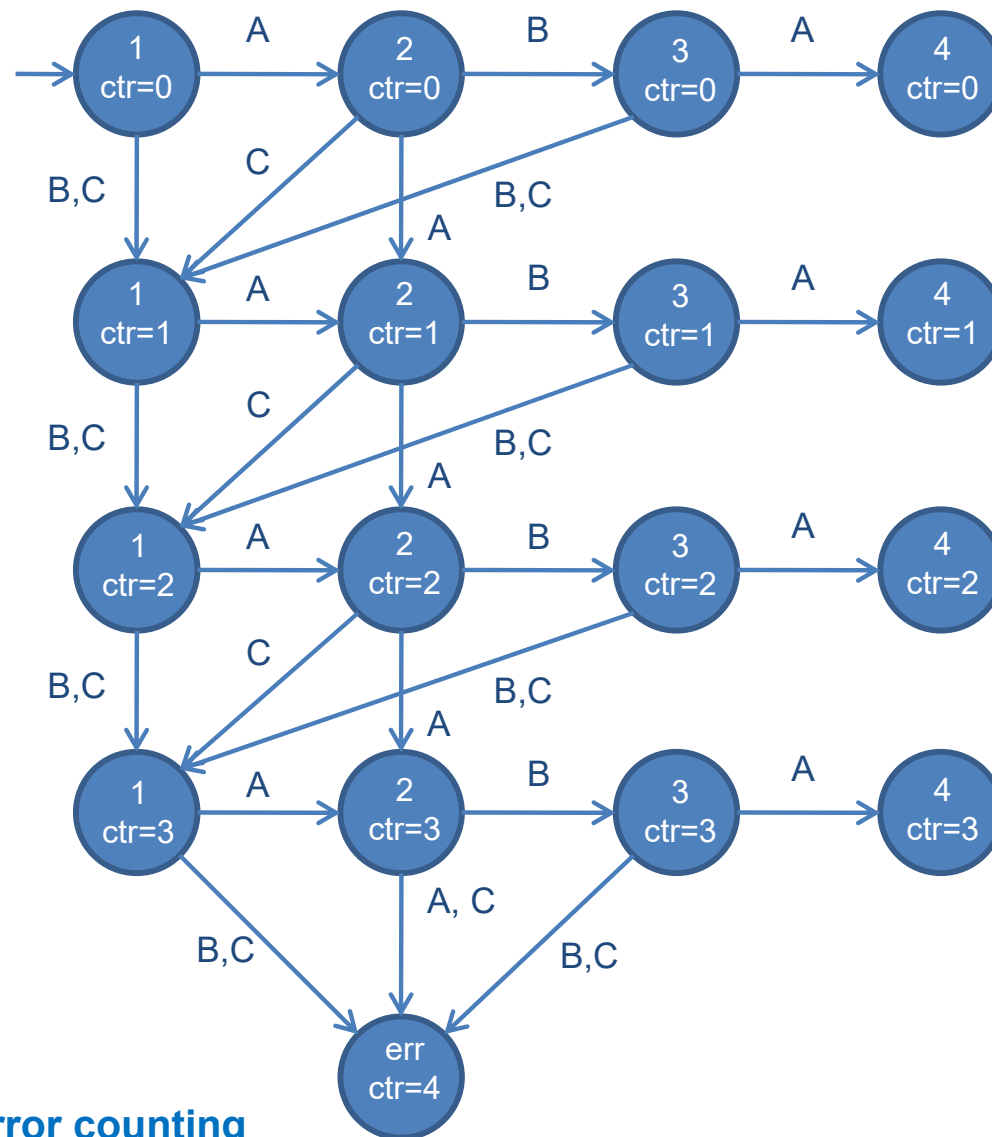


No more than 3 mistakes !!!

The digicode with guarded transitions



Unfolding



The digicode with error counting (Unfolded automaton)

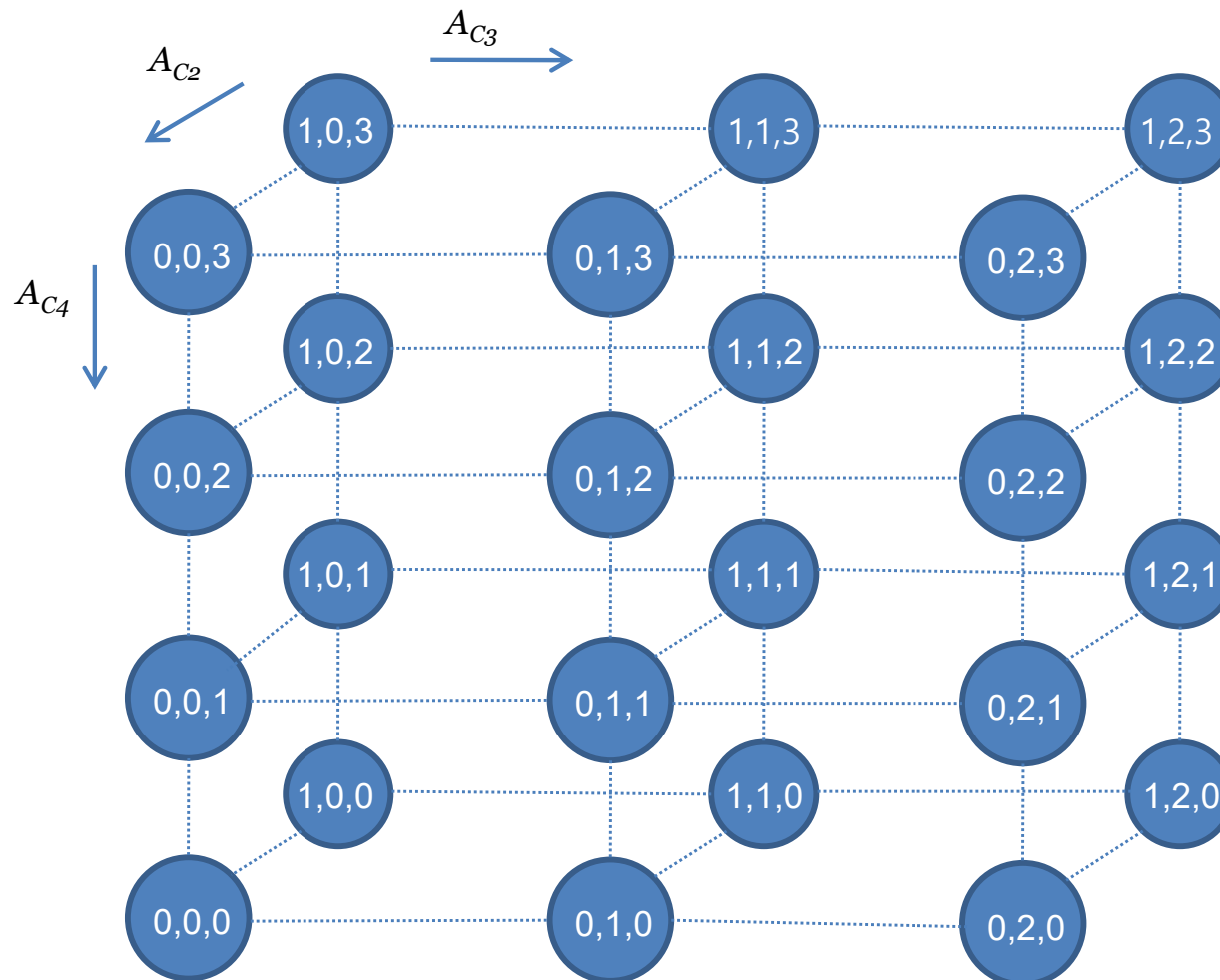
1.5 Synchronized Product

- 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)

 - We need to construct it with small component automata.
 - Two composition ways
 - **With synchronization (= Synchronized product)**
 - **Without synchronization (= Cartesian product)**

- An example without synchronization
 - 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



$$2 \cdot 3 \cdot 4 = 24 \text{ states}$$

$$3 \cdot 3 \cdot 3 - 1 = 26 \text{ transitions per a state (Inc, Dec, -)}$$

$$\rightarrow 24 \cdot 26 = 624 \text{ transitions}$$

- An example with synchronization
 - There are 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**

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

- $A = \langle Q, E, T, q_0, l \rangle$

- $Q = Q_1 \times Q_2 \times \dots \times Q_n$

- $E = \prod_{1 \leq i \leq n} (E_i \cup \{-\})$

- $T = \left[\begin{array}{l} ((q_1, \dots, q_n), (e_1, \dots, e_n), (q'_1, \dots, q'_n)) \mid \text{for all } i, \\ (e_i = '-' \text{ and } q'_i = q_i) \text{ or } (e_i \neq '-' \text{ and } (q_i, e_i, q'_i) \in T_i) \end{array} \right]$

- $q_0 = (q_{0,1}, \dots, q_{0,n})$

- $l((q_1, \dots, q_n)) = \bigcup_{1 \leq i \leq n} l_i(q_i)$

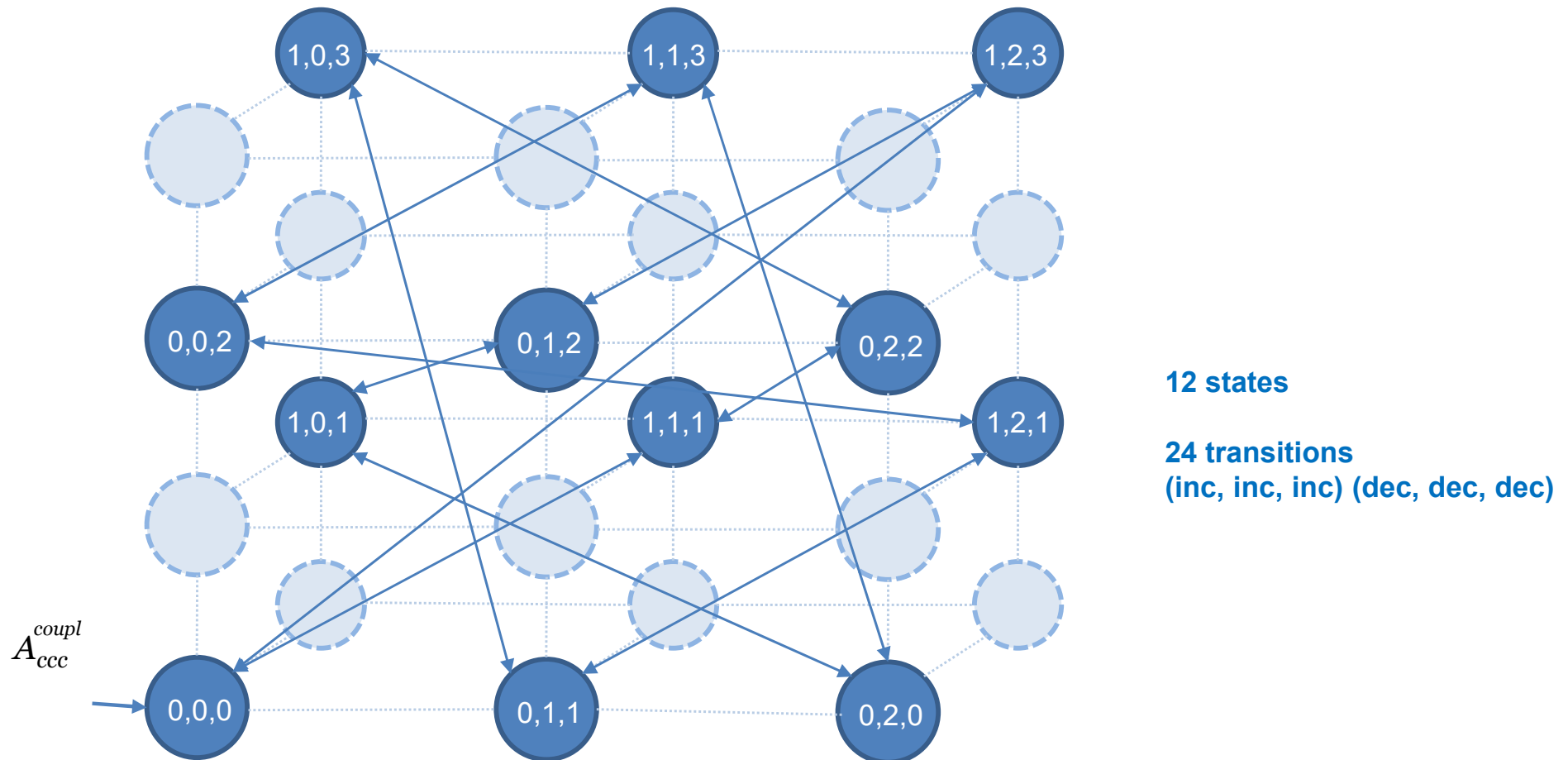
- **Sync** $\subseteq \prod_{1 \leq i \leq n} (E_i \cup \{-\})$: Synchronized set

- An example with synchronization

- Allowing only “inc, inc, inc” and “dec, dec, dec” (24*2=48 transitions)
→ Strongly coupled version of modular counters

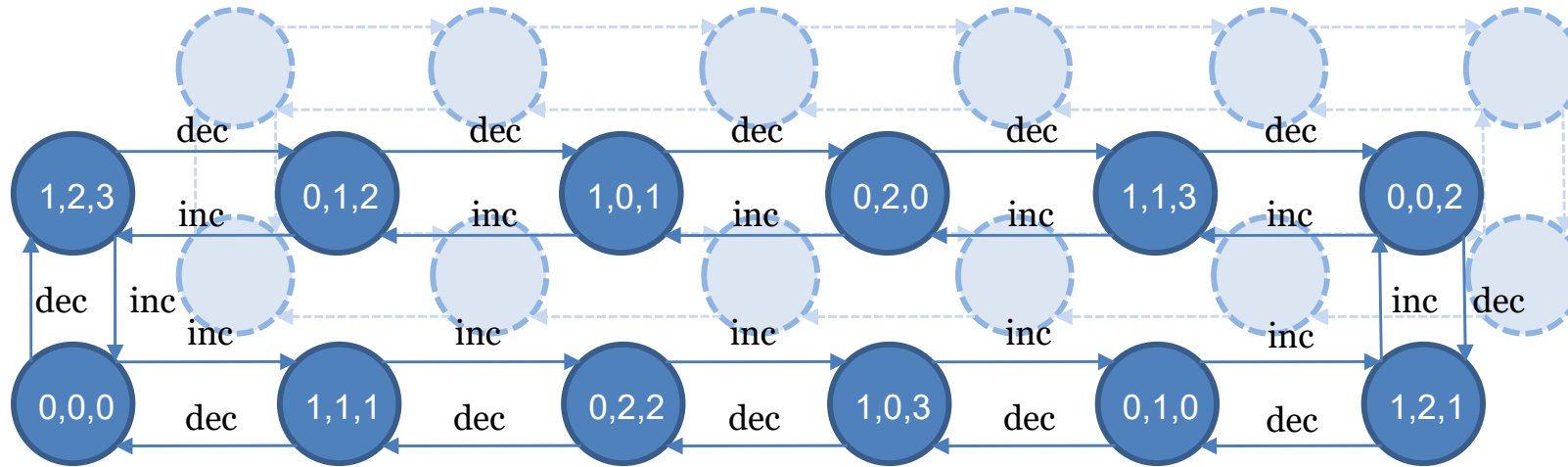
- $Sync = \{ (inc, inc, inc), (dec, dec, dec) \}$

- $$T = \left[\begin{array}{l} ((q_1, \dots, q_n), (e_1, \dots, e_n), (q'_1, \dots, q'_n)) \mid (e_1, \dots, e_n) \in Sync \\ (e_i = '-' \text{ and } q'_i = q_i) \text{ or } (e_i \neq '-' \text{ and } (q_i, e_i, q'_i) \in T_i) \end{array} \right]$$



- **Reachable states**

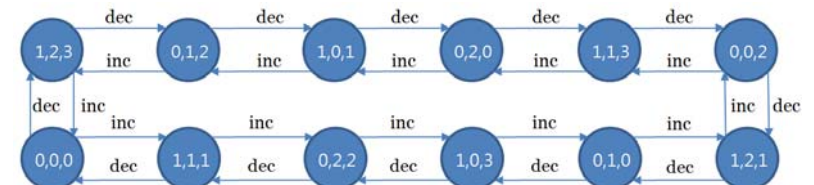
- Reachability depends on the synchronization constraints



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

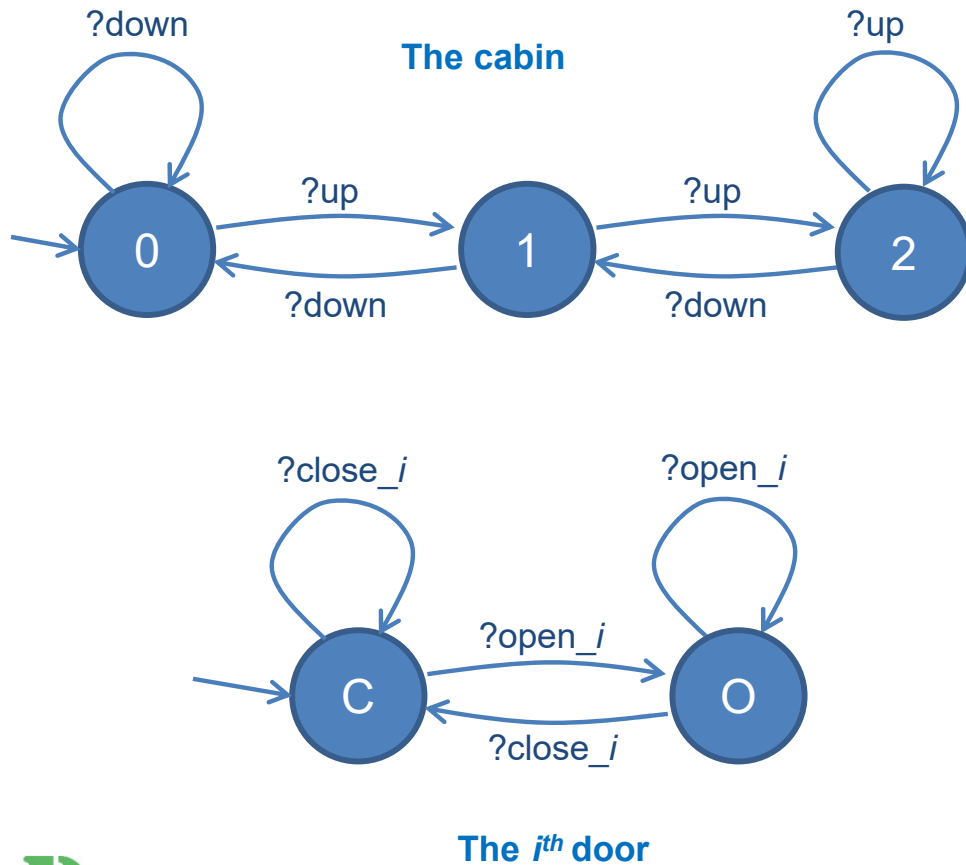
- Message passing framework
 - A special case of synchronized product
 - **!m** : Emitting a message
 - **?m** : Reception of the message

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

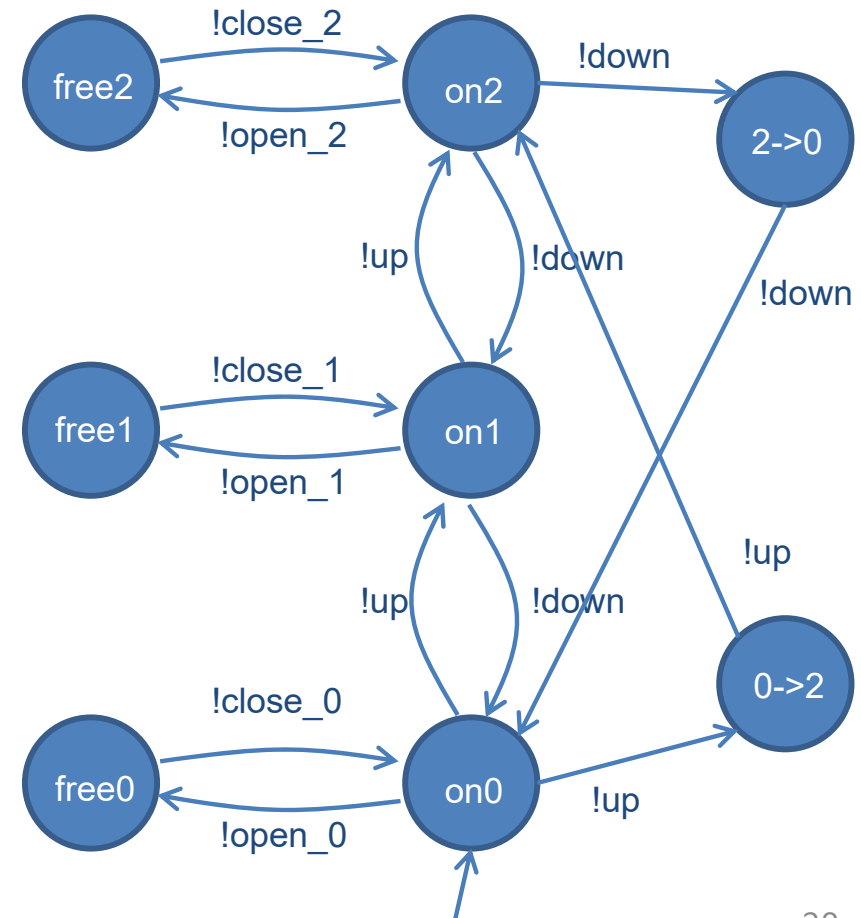
 - Synchronous communication
 - Control/command system
 - Asynchronous communication
 - Communication protocol (using channel/buffer)

• Smallish elevator

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



The controller



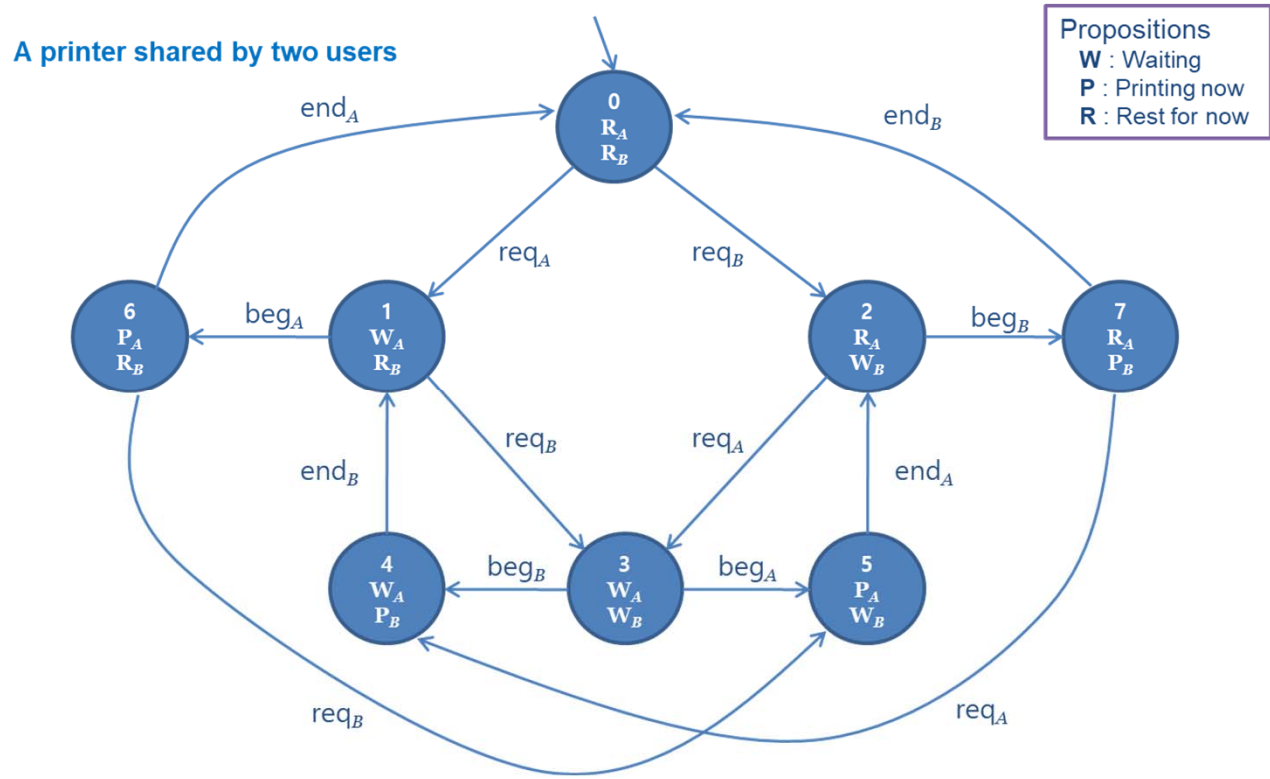
- An automaton for the smallish elevator example
 - Obtained as the synchronized product of the 5 automata
 - (door 0, door 1, door 2, cabin, controller)
 - $Sync = \{ (?open_0, -, -, -, !open_0), (?close_0, -, -, -, !close_0), (-, ?open_1, -, -, !open_1), (-, ?close_1, -, -, !close_1), (-, -, ?open_2, -, !open_2), (-, -, ?close_2, -, !close_2), (-, -, -, ?down, !down), (-, -, -, ?up, !up) \}$

- **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.

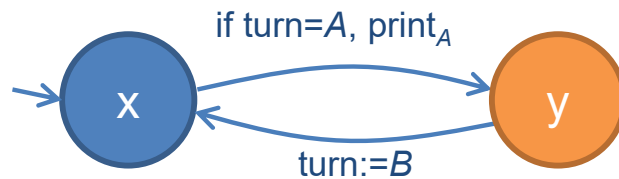
1.7 Synchronization by Shared Variables

- 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.

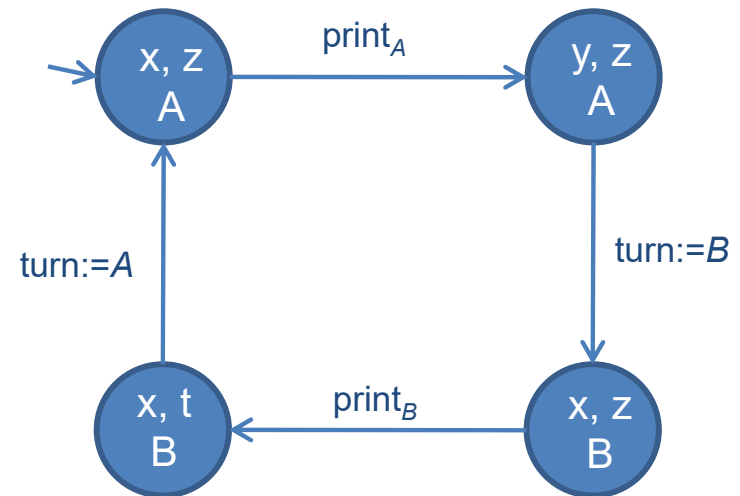
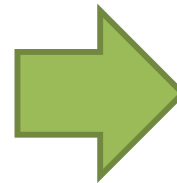
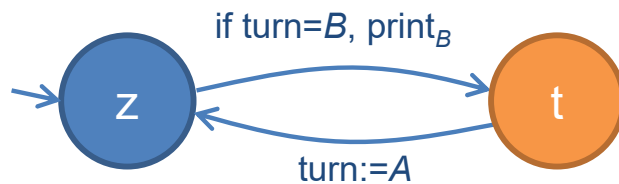


- The printer manager synchronized with a shared variable
 - Shared variable: *turn*
- Fairness property: “*Any print request is ultimately satisfied.*”
 - No state of the form $(y, t, -)$ is reachable.
 - TRUE in the model
 - But this model forbids either user from printing twice in a row.

The user A

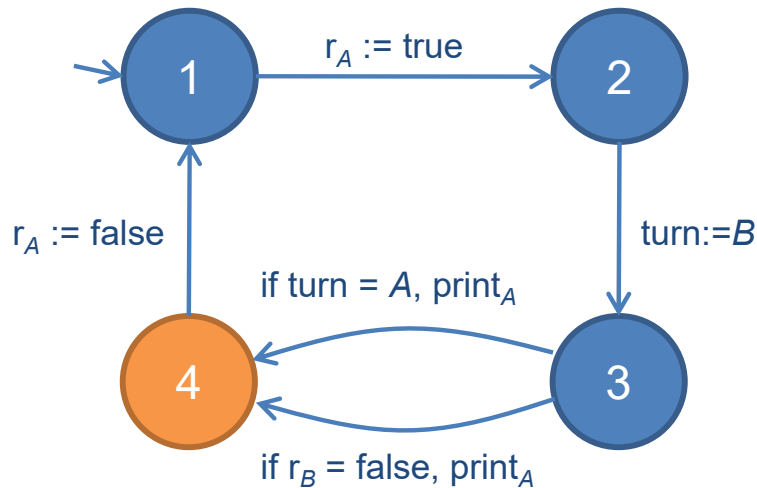


The user B

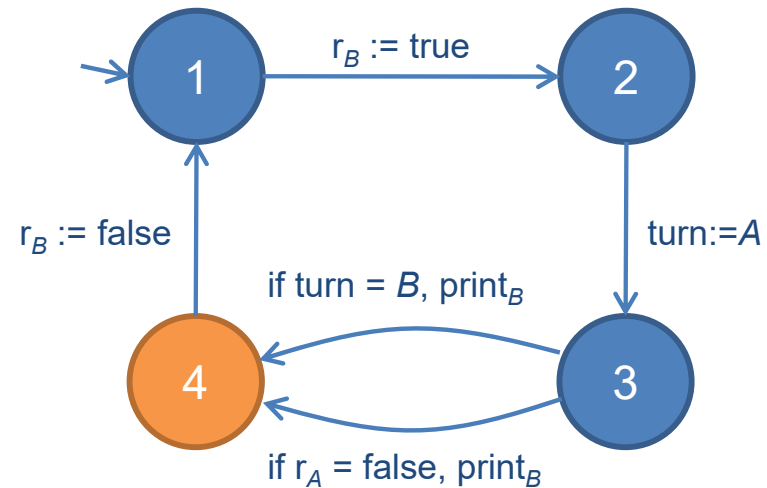


- Printer manager : A complete version with 3 variables [by Peterson]
 - r_A : a request from user A
 - r_B : a request from user B
 - $turn$: to settle conflicts
 - Satisfies all our properties.

The user A



The user B



$$A_{A \times B} = \langle Q, E, T, q_0, l \rangle$$

$$- Q = A \times B \times r_A \times r_B \times turn$$

$$4 \times 4 \times 2 \times 2 \times 2 = 128 \text{ states (only 128 reachable states)}$$

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
 - In a **first order logic**,
 - $\forall t, \forall n (app(n, t) \Rightarrow \exists t' > t : serv(n, t'))$
 - $\forall t, \forall t' > t, \forall n, \left[\begin{array}{l} (app(n, t) \wedge H(t') \neq n \wedge \exists t_{trav} : \\ t \leq t_{trav} \leq t' \wedge H(t_{trav}) = n) \\ \Rightarrow (\exists t_{serv} : t \leq t_{serv} \leq t' \wedge serv(n, t_{serv})) \end{array} \right]$
 - 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
 - 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

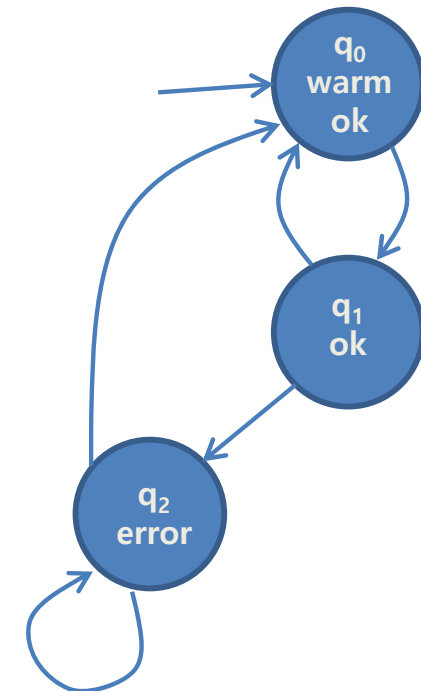
- CTL*
 - serves to formally state the properties concerned with the execution of a system
 - variants (CTL, PLTL, LTL)
 - 6 characteristics

1. Atomic Propositions

- warm, ok, error

2. Proposition Formula

- using Boolean combinators
- true, false, \neg , \vee , \wedge , \Rightarrow (if then), \Leftrightarrow (if and only if)
- $\text{error} \Rightarrow \neg \text{warm}$
(if error then not warm)



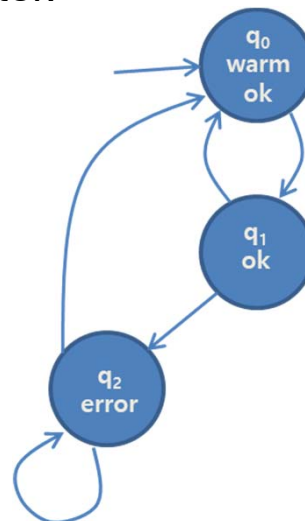
$\sigma_1 : (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow \dots$
 $\sigma_2 : (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow (q_2: \text{error}) \rightarrow (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow \dots$
 $\sigma_3 : (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow (q_2: \text{error}) \rightarrow (q_2: \text{error}) \rightarrow (q_2: \text{error}) \rightarrow \dots$

3. Temporal combinators

- about the sequencing of states along an execution
- X : next state
- F : a future state $\sigma_2 : (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow (q_2: \text{error}) \rightarrow (q_0: \text{warm, ok}) \rightarrow (q_1: \text{ok}) \rightarrow \dots$
- G : all the future states

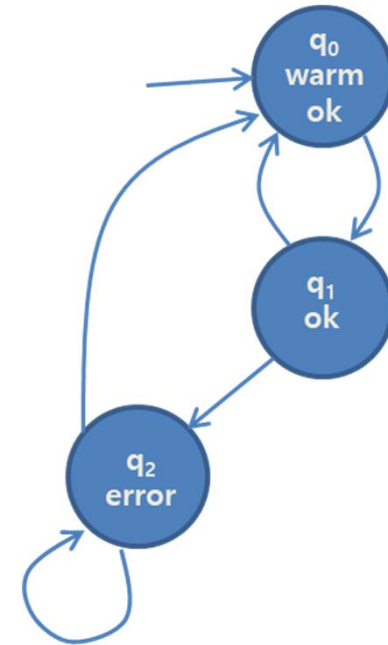
- $X P$: the next state satisfies P
- $F P$: a future state satisfies P without specifying which state
 $\rightarrow P$ will hold some day (at least once)
- $G P$: all future states will satisfy P
 $\rightarrow P$ will always be
- $alert \Rightarrow F halt$: if we are currently in a state of *alert*, 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
- $GF \phi$: always there will some day be a state such that ϕ , ϕ is satisfied infinitely often along the execution considered
- $FG \phi$: all the time from a certain time onward, at each time instant, possibly excluding a finite number of instants
- $GF \textit{warm} \vee FG \textit{error}$

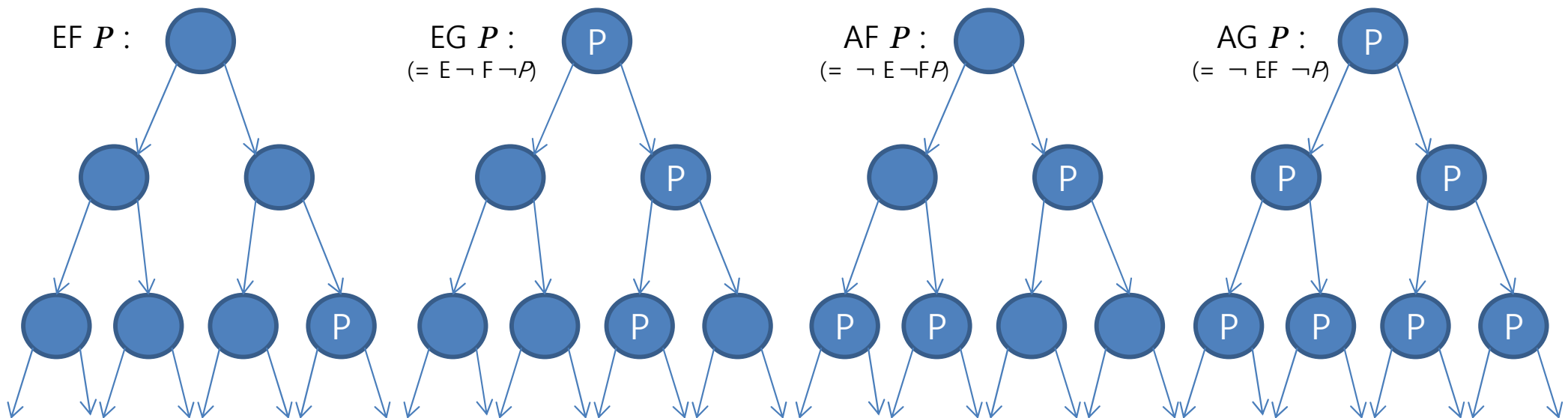


5. U combinator

- for until
- $\phi_1 U \phi_2$: ϕ_1 is verified until ϕ_2 is verified
 ϕ_2 will be verified some day, and ϕ_1 will hold in the meantime
- $G (\textit{alert} \Rightarrow (\textit{alarm} U \textit{halt}))$: starting from a state of alert, the alarm remains activated until the halt state is eventually and inexorably reached.
- $F \phi \equiv \textit{true} U \phi$
- $\phi_1 W \phi_2 \equiv (\phi_1 U \phi_2) \vee G \phi_1$: weak until

6. Path quantifier

- $A \phi$: all the executions out of the current state satisfy property ϕ
- $E \phi$: 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



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.

- $\phi, \Psi ::= P1 \mid P2 \mid \dots$ (atomic proposition)
- | $\neg\phi \mid \phi \wedge \Psi \mid \phi \Rightarrow \Psi \mid \dots$ (boolean combinators)
- | $X\phi \mid F\phi \mid G\phi \mid \phi U\Psi \mid \dots$ (temporal combinators)
- | $E\phi \mid A\phi$ (path quantifiers)

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. $A = \langle Q, T, q_0, l \rangle$, $T \subseteq Q \times Q$

- **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 $\sigma, 0 \models \phi$ 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 $\sigma, i \models \psi$,
$\sigma, i \models X\phi$	iff $i < \sigma $ and $\sigma, i + 1 \models \phi$,
$\sigma, i \models F\phi$	iff there exists j such that $i \leq j \leq \sigma $ and $\sigma, j \models \phi$,
$\sigma, i \models G\phi$	iff for all j such that $i \leq j \leq \sigma $, we have $\sigma, j \models \phi$,
$\sigma, i \models \phi U \psi$	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 $\sigma, k \models \phi$,
$\sigma, i \models E\phi$	iff there exists a σ' such that $\sigma(0) \dots \sigma(i) = \sigma'(0) \dots \sigma'(i)$ and $\sigma', i \models \phi$,
$\sigma, i \models A\phi$	iff for all σ' such that $\sigma(0) \dots \sigma(i) = \sigma'(0) \dots \sigma'(i)$, we have $\sigma', i \models \phi$.

Semantics of CTL*

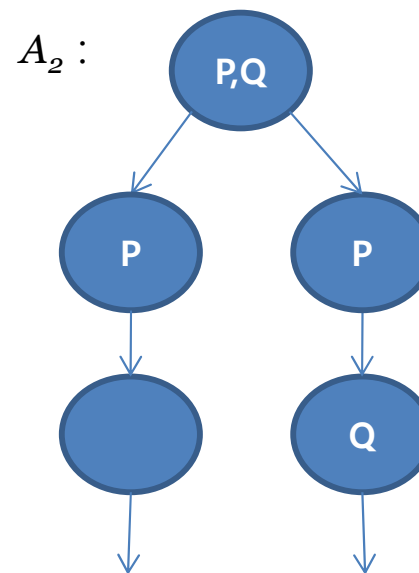
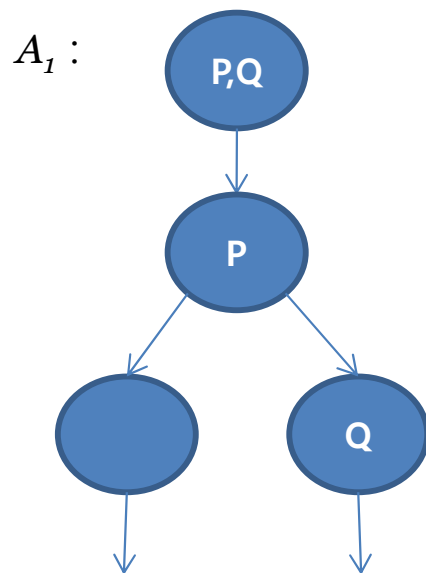
CTL*

- 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*

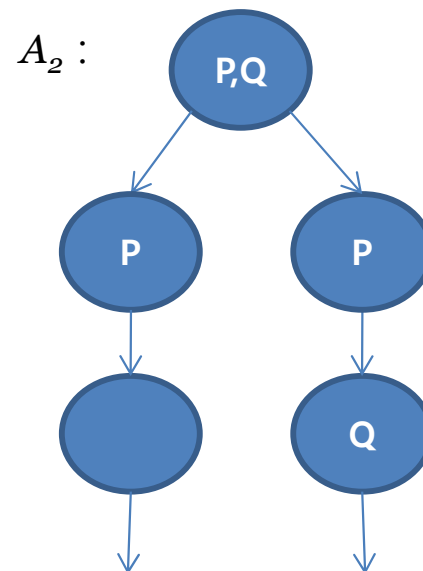
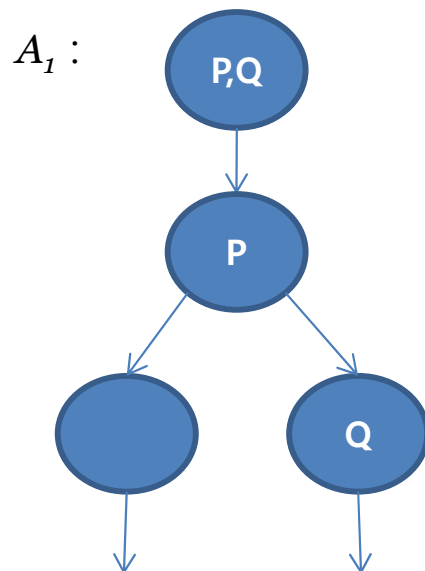
- **PLTL**
 - No path quantifiers (A and E)
 - Linear time logic \rightarrow Path formula
 - Ex. PLTL cannot distinguish A_1 from A_2



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 A_1 and A_2



$$A_1, q_0 \models AX (EXQ \wedge EX\neg Q)$$

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

- 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, we have no choice
 - SMV : CTL / PLTL
 - SPIN : PLTL
 - VIS : CTL / PLTL

 - No model checking tool for CTL*

3. Model Checking

3. Model Checking

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

- Organization
 - 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

- Complexity of the algorithm
 - Model checking “ does $A, q_0 \models \phi$? ” for a CTL formula ϕ
 - can be solved in time $O(|A| \times |\phi|)$
 - $O(|A|)$: 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

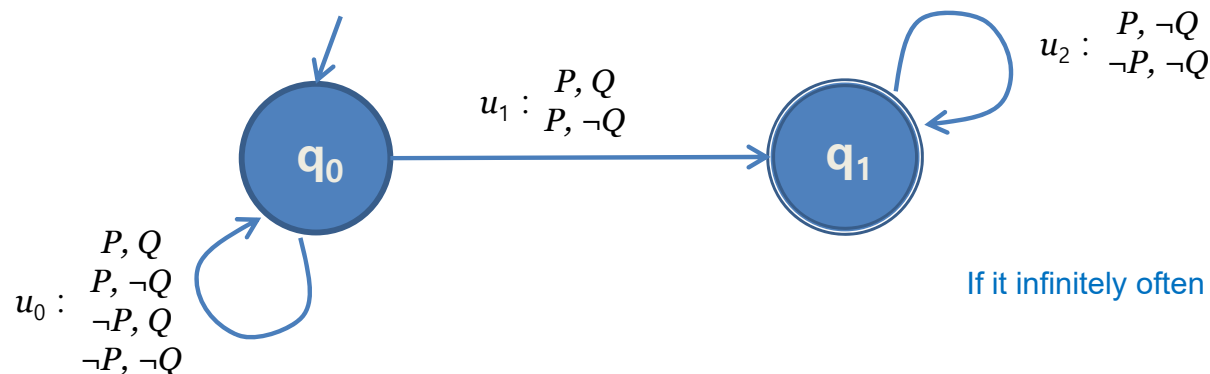
- Model checking algorithm for PLTL
 - Developed in 1980s, but too technical to cover in this course
 - Not possible to rely on marking the automaton states, since PLTL uses path formulas.
 - A finite automaton will generally give rise to infinitely many different executions, themselves often infinite 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 $B_{\neg\phi} \rightarrow A \otimes B_{\neg\phi}$
- Finally reduces to “ **Is the language recognized by $A \otimes B_{\neg\phi}$ empty ?** ”

- A simple example

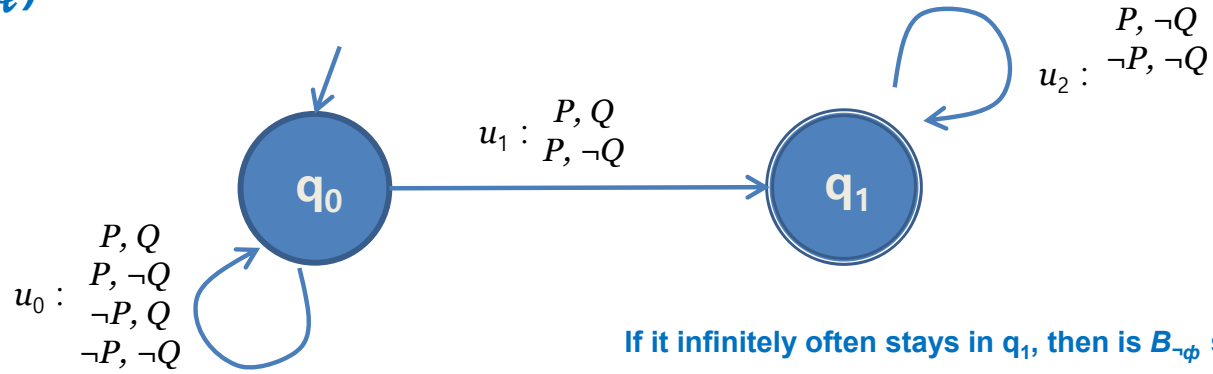
- $\phi : G(P \Rightarrow XF Q)$: Any occurrence of P must be followed (later) by an occurrence of Q
- $B_{\neg\phi}$: There exists an occurrence of P after which we will never again encounter Q



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

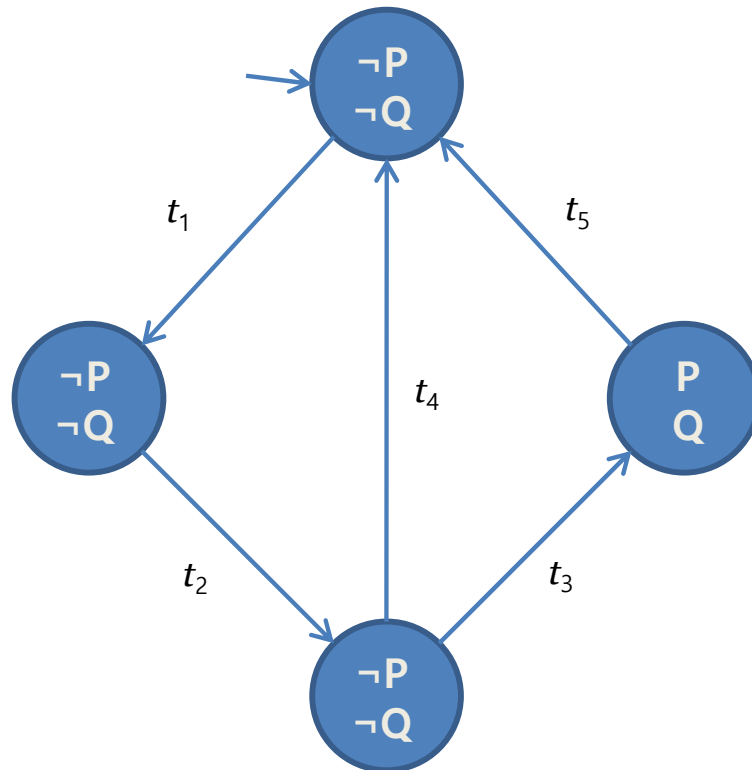
$\phi : G(P \Rightarrow XF Q)$

$B_{\neg\phi} :$



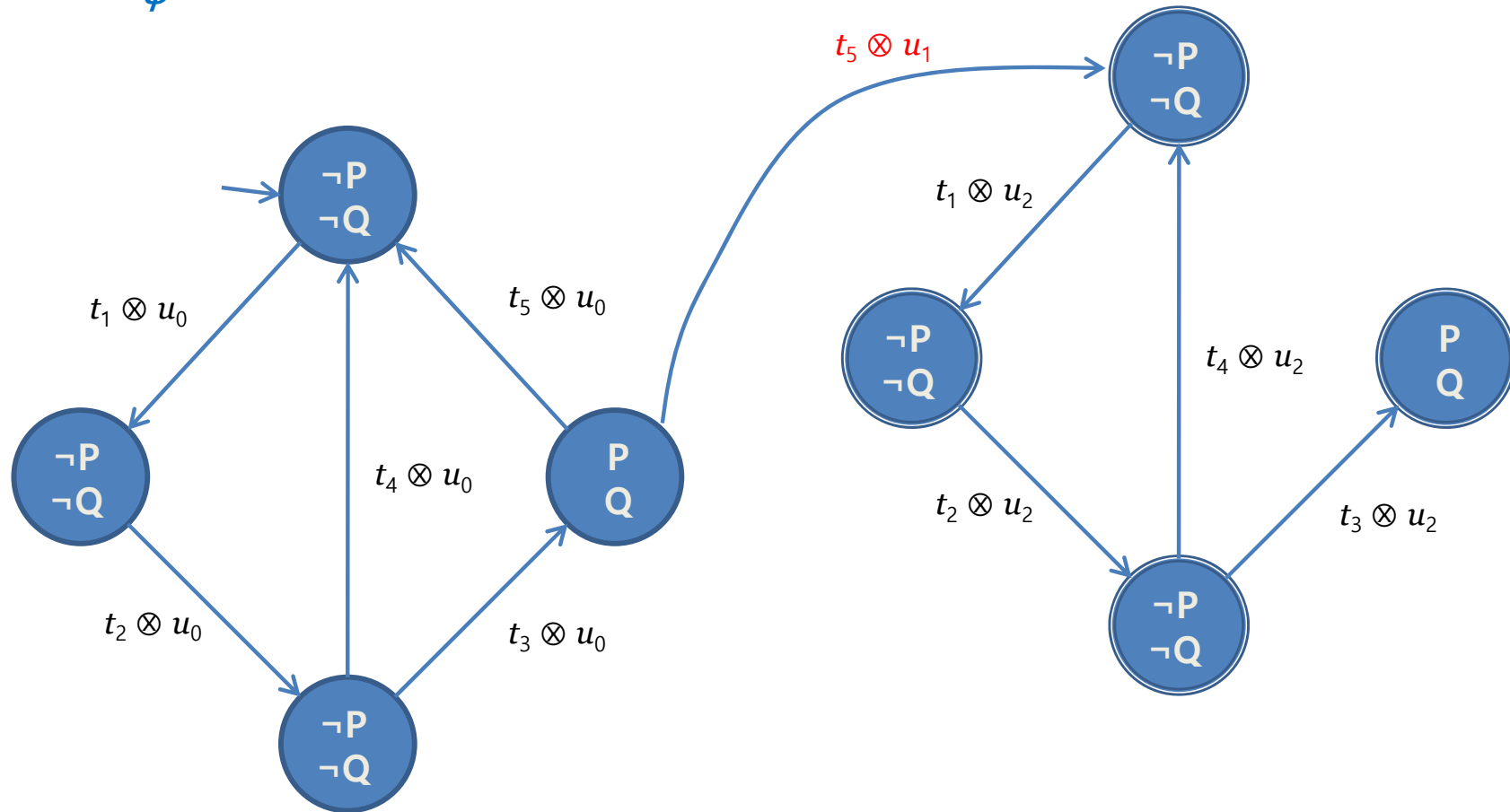
If it infinitely often stays in q_1 , then is $B_{\neg\phi}$ satisfied.

$A :$



" 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.

→ $A \not\equiv \phi$

- 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
 - 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)

Patterns of Temporal Properties

- Writing the temporal logic formulas expressing desired system properties is important but difficult.
 - No silver bullet
 - No automatic generation

- 4 classification (categories/patterns) according to verification goals
 - **Reachability** property
 - Some particular situation can be reached.
 - **Safety** property
 - Under certain condition, something never occurs.
 - **Liveness** property
 - Under certain condition, something will ultimately occur.
 - **Fairness** property
 - Under certain condition, something will (or not) occur infinitely often.

 - **Deadlock freeness**

4. Properties

Reachability Properties

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 “

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 $n < 0$ “
 - \neg EF ($n < 0$)
 - (R4) “ We cannot reach the crash state “
 - \neg EF crash
 - AG \neg crash
 - “ Along every path, at any time, \neg crash ”
 - (R5) “ We can enter the critical section without traversing $n = 0$ “
 - E ($n \neq 0$) U crit_sec
 - “ There exists a path along which $n \neq 0$ 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 \rightarrow add their successors \rightarrow until saturation
 - Difficulty:
 - Potential explosion of the set constructed

- Backward chaining
 - from target states \rightarrow add immediate predecessors \rightarrow until saturation, then, test whether some initial states are in there
 - Difficulties:
 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

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
 - \neg safety property = reachability property
 - \neg reachability property = safety property

7.1 Safety Properties in Temporal Logic

- **AG $\neg \phi$**
 - “ ϕ never occurs. “
 - (S1) “ Both processes will never be in their critical sections simultaneously ”
 - $AG \neg(\text{crit_sec1} \wedge \text{crit_sec2})$
 - (S2) “ Memory overflow will never occur ”
 - $AG \neg\text{overflow}$
 - (S3) “ The situation ... is impossible “
 - $AG \neg\text{situation}$
 - (S4) “ As long as the key is not in the ignition position, the car won't start “
 - $A (\neg\text{start} W \text{key})$ ← using weak until : it is a safety property
 - $A (\neg\text{start} U \text{key})$ ← using strong until : Not a safety property!

7.2 A Formal Definition

- Syntactic characterization
 - Safety properties can be written in the form $AG \phi^-$
 - 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})

- $AG \phi^-$ in practice
 - (S1) $AG \neg(\text{crit_sec}_1 \wedge \text{crit_sec}_2)$
 - $\neg(\text{crit_sec}_1 \wedge \text{crit_sec}_2)$ is a ϕ^-
 - (S4) $A \neg \text{start } W \text{ key}$
 - Can be rewritten in the form: $AG (\text{start} \Rightarrow F^{-1} \text{ key})$
 - " It is always true (AG) that if the car starts, then (\Rightarrow) the key was inserted beforehand (F^{-1}). "

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
 - $\neg EF (\text{crit_in}_1 \wedge \text{crit_in}_2) = AG \phi^-$
 - $\neg(\text{crit_in}_1 \wedge \text{crit_in}_2)$ is a present formula
- Safety without past
 - $A (\neg \text{start } W \text{ key})$ vs. $AG (\text{start} \Rightarrow F^{-1} \text{ key})$
 - 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)
 - $AG (\phi \Rightarrow F^{-1} \psi) \equiv A (\neg \phi W \psi)$, but not easy.
 2. History variable method (section 7.4)

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)

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
 - $P U Q \equiv F Q \wedge (P W Q)$
 (liveness) (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” is more general than “concrete”.
 - “Abstract” is more efficient than “concrete”.
 - “Abstract” and “concrete” are not contradictory.

8.3 Liveness in the Model and 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 :



- 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
 - $AF (E P U Q) \rightarrow A (\Phi_v \Rightarrow FE (\Phi_v \wedge P U Q))$

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 \rightarrow TCTL !!!
- (BL1) “ The program terminates in less than ten seconds “
 - $AF_{<10s} \text{ end}$ \leftarrow bounded liveness property
 - $AG (\neg \text{end} \Rightarrow F^{-1}_{<10s} \text{ start})$ \leftarrow safety property
- (BL2) “ Any request is satisfied in less than five minutes “
 - $AG (\text{req} \Rightarrow AF_{<5m} \text{ sat})$ \leftarrow bounded liveness property
 - $AG (\neg(F^{-1}_{=5m} \text{ req} \wedge G^{-1}_{\leq 5m} \neg \text{sat}))$ \leftarrow safety property

Deadlock-Freeness

9. Deadlock-Freeness

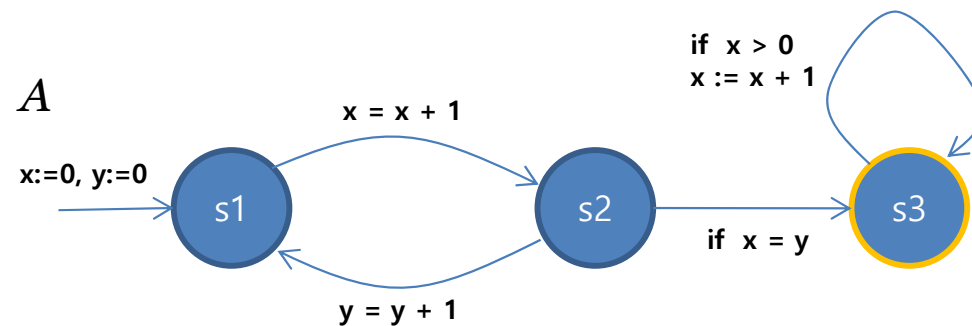
- **Deadlock-freeness**
 - A special property, “ *The system can never be in a situation on which no progress is possible.* ”
 - Correct property relevant for systems that are supposed to run indefinitely
 - A set of properly identified final states will be required to be deadlock-free.

9.1 Safety? Liveness?

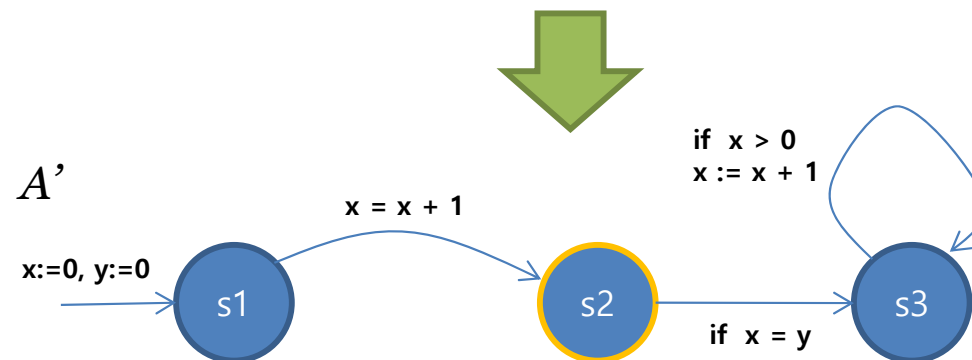
- **AG EX true**
 - “ Whatever the state reached may be (AG), there will exist an immediate successor state (EX true) ”
 - Not the form of $AG\phi^1$
 - Deadlock-free is not a safety property.
 - Can be verified if the model checker can handle AG EX true.

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,

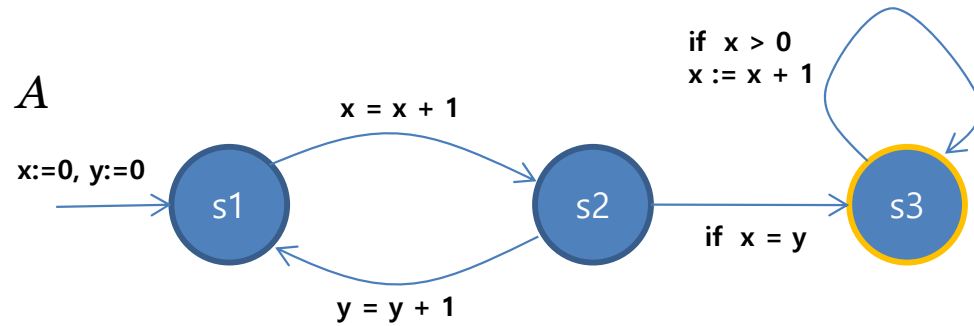


$AG\ EX\ true \rightarrow$ hold! (liveness property)
 $AG\ \neg(s3 \wedge x \leq 0) \rightarrow$ hold! (safety property)

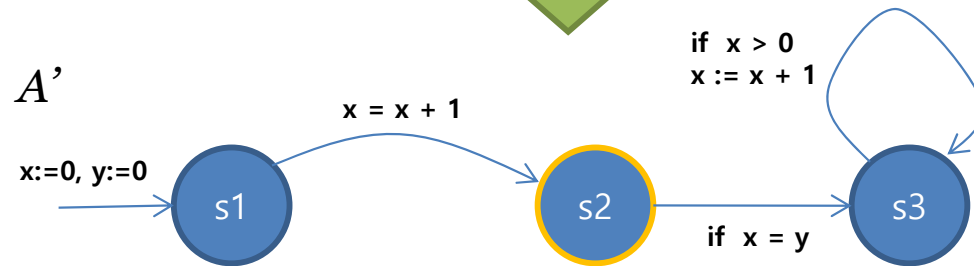


$AG\ EX\ true \rightarrow$ not hold! (liveness property)
 $AG\ \neg(s3 \wedge x \leq 0) \rightarrow$ hold! (safety property)

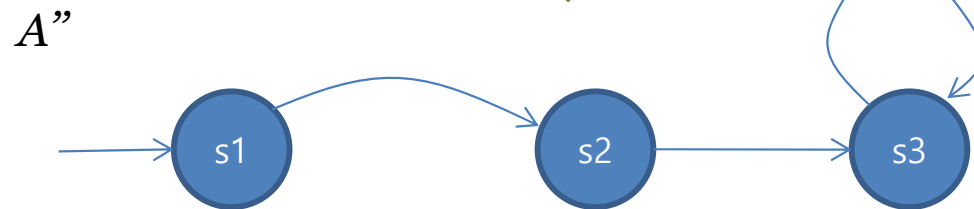
9.3 Beware of Abstractions!



Deadlock-free



Deadlock



Deadlock-free

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

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 gate_raised)
 - (F2) “ If access to a critical section is infinitely often requested, then access will be granted infinitely often. ”
→ A (GF crit_req ⇒ FG crit_in)
- FCTL or ECTL+
 - CTL + fairness
 - $O(|A| \times |\phi|^2)$
 - Many tools (like SMV) considers the fairness hypotheses as part of model rather 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 !!!
 - $A (\text{fairness hypothesis } \neg \text{loss} \Rightarrow G (\text{liveness property } \text{emitted} \Rightarrow F \text{ received}))$
 - **Repeated liveness property** : “ If infinitely many messages are emitted, then infinitely many messages will be transmitted.”
 - $A (\text{fairness hypothesis } \neg \text{loss} \Rightarrow (\text{repeated liveness property } GF \text{ emitted} \Rightarrow GF \text{ received}))$

10.4 Strong Fairness and Weak Fairness

- Fairness property
 - “If P is continually requested, then P will be granted (infinitely often).”
 - Weak fairness : without interruption
 - Strong fairness : possibly with interruption
 - No difference when using them for model checking of finite systems

- **Weak fairness**
 - Assume that P is requested without interruption
 - $(FG \text{ request_P}) \Rightarrow F P$
 - $(FG \text{ request_P}) \Rightarrow GF P$

- **Strong fairness**
 - Assume that P is requested in an infinitely repeated manner, possibly with interruptions
 - $(GF \text{ request_P}) \Rightarrow F P$
 - $(GF \text{ request_P}) \Rightarrow GF P$

10.5 Fairness in the Model or in the Property?

- **The best way is**
 - **Model = automaton + fairness hypothesis**
 - Pros: Fairness hypothesis can change independently from the automata model.
 - Ex. SMV model checker

- Symbolic Model Checking
- Timed Automata

FORMAL VERIFICATION : ADVANCED

5. 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(Binary Decision Diagram)** are used to represent the state variables
 - Represent very large sets of states concisely, as if they were in bulk.

- 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.

- Organization
 - 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 $\text{Sat}(\phi)$
 - $A = \langle Q, T, \dots \rangle$
 - $\text{Pre}(S)$: immediate predecessors of the states belonging to S in Q
 - **$\text{Sat}(\phi)$** : set of states of A which satisfy ϕ
 - ψ is the sub-formulas of ϕ
 - $\text{Sat}(\neg\psi) = Q \setminus \text{Sat}(\psi)$
 - $\text{Sat}(\psi \wedge \psi') = \text{Sat}(\psi) \cap \text{Sat}(\psi')$
 - $\text{Sat}(\text{EX } \psi) = \text{Pre}(\text{Sat}(\psi))$
 - $\text{Sat}(\text{AX } \psi) = Q \setminus \text{Pre}(Q \setminus \text{Sat}(\psi))$
 - $\text{Sat}(\text{EF } \psi) = \text{Pre}^*(\text{Sat}(\psi))$
 - ... (others are defined in a similar way)
 - The algorithms in Section 3.1 is an particular implementation of $\text{Sat}(\phi)$
 - Hence, **$\text{Sat}(\phi)$** is an explicit representation of the state sets.

```

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

```

- 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.

- 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)

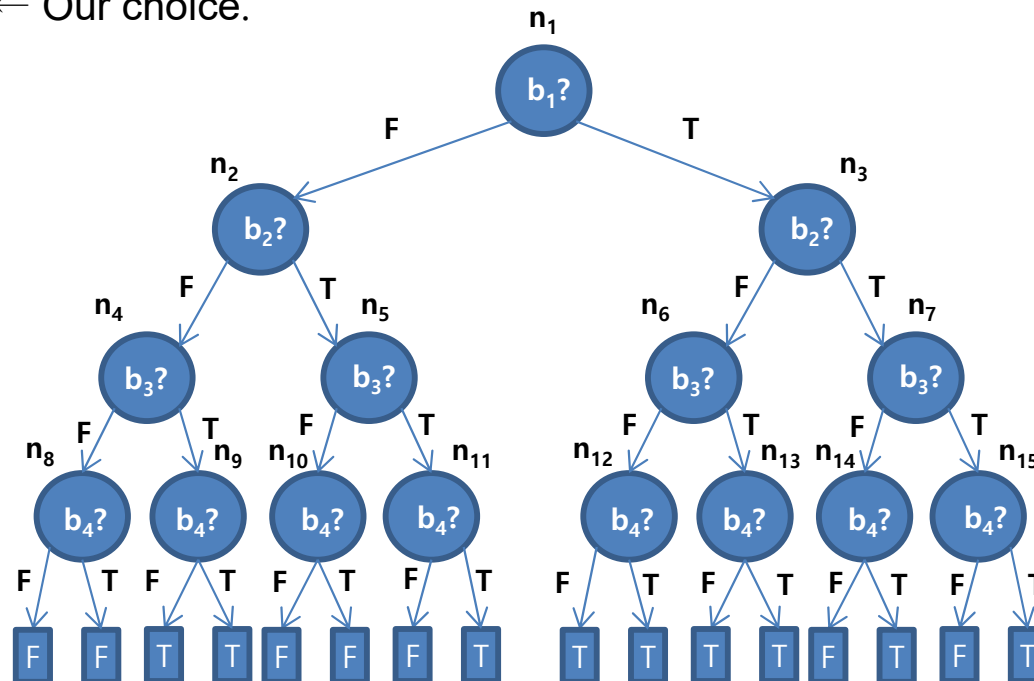
- **BDD**
 - A particular data structure very commonly used for representing states sets symbolically
 - Proposed in 1980s ~ early in 1990s
 - Make possible the verification of the system which cannot represent explicitly.
 - Advantages:
 - Efficiency
 - Simplicity
 - Easy Adaptation
 - Generality

- BDD structure example

- n boolean variables x_1, x_2, \dots, x_n associated with a tuple $\langle b_1, b_2, \dots, b_n \rangle$
 - 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:

- $S = \{ \langle F, F, T, F \rangle, \langle F, F, T, T \rangle, \dots \}$
 - $S = (b_1 \vee b_2) \wedge (b_3 \Rightarrow b_4)$
 - $S = (b_1 \wedge \neg b_2) \vee (b_1 \wedge b_4) \vee (b_3 \wedge \neg b_2) \vee (b_3 \wedge b_4) \leftarrow$ DNF
 - ...
 - Decision Tree** \leftarrow Our choice.



- Decision tree reduction

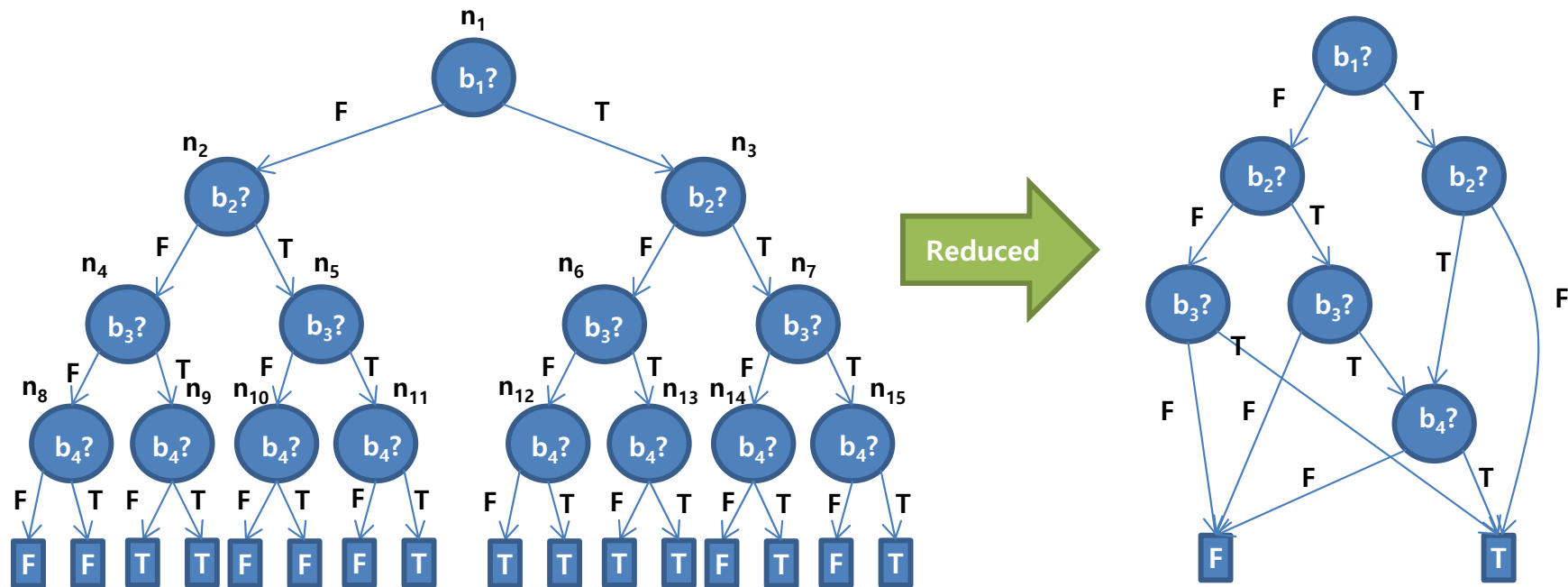
- A BDD is a reduced decision tree.

- 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

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

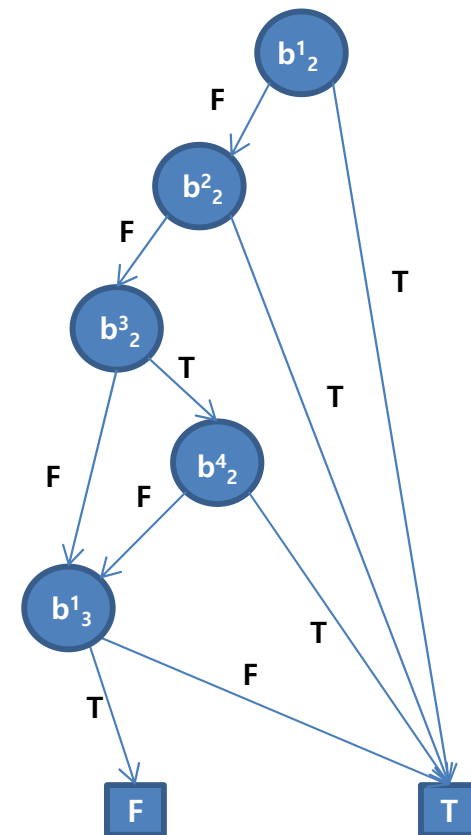
- Before applying BDDs to symbolic model checking, we need to restate
 - Representing the **states** by BDDs
 - Representing **transitions** by BDDs

- Representing the states by BDDs

- Consider an automaton A with

- $Q = \{q_0, \dots, q_6\} \rightarrow b^1_1, b^2_1, b^3_1$
- var digit:0..9 $\rightarrow b^1_2, b^2_2, b^3_2, b^4_2$
- var ready:bool $\rightarrow b^1_3$
- $\langle b^1_1, b^2_1, b^3_1, b^1_2, b^2_2, b^3_2, b^4_2, b^1_3 \rangle$
- $\langle F, T, T, T, F, F, F, F \rangle = \langle q_3, 8, F \rangle$

- Let's represent $Sat(ready \Rightarrow (digit > 2))$
 - States $\langle q, k, b \rangle$ such that if $b = T$ and $k > 2$
 - $ready \Rightarrow (digit > 2) \equiv \neg ready \vee (digit > 2)$

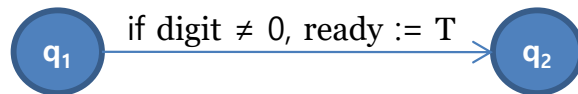


- Representing transitions by BDDs

- The same idea is applied.

- $\langle q_3, 8, F \rangle \rightarrow \langle q_5, 0, F \rangle : \langle F, T, T, T, F, F, F, F, T, F, T, F, F, F, F, F \rangle$

- For example,



- $(\langle q, k, b \rangle, \langle q', k', b' \rangle)$

- $q = q_1, k \neq 0, q' = q_2, k' = k, b' = T$

- $(\neg b^1_1 \wedge \neg b^2_1 \wedge b^3_1)$

- $\wedge (b^1_2 \vee b^2_2 \vee b^3_2 \vee b^4_2)$

- $\wedge (\neg b'^1_1 \wedge b'^2_1 \wedge \neg b'^3_1)$

- $\wedge (b'^1_2 \Leftrightarrow b^1_2 \wedge b'^2_2 \Leftrightarrow b^2_2 \wedge b'^3_2 \Leftrightarrow b^3_2 \wedge b'^4_2 \Leftrightarrow b^4_2)$

- $\wedge b'^1_3$

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.

6. 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”

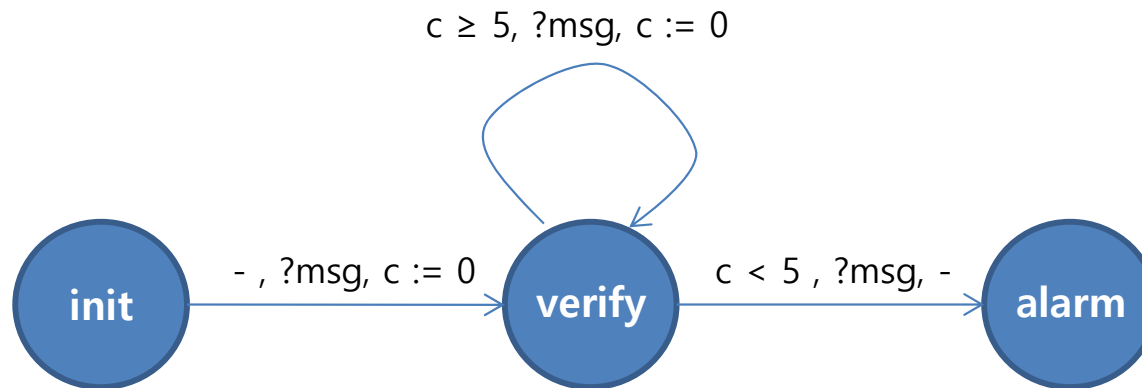
- Timed Automata
 - Proposed by Alur and Dill in 1994.
 - An answer to this “real-time” needs

- Organization
 - 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**

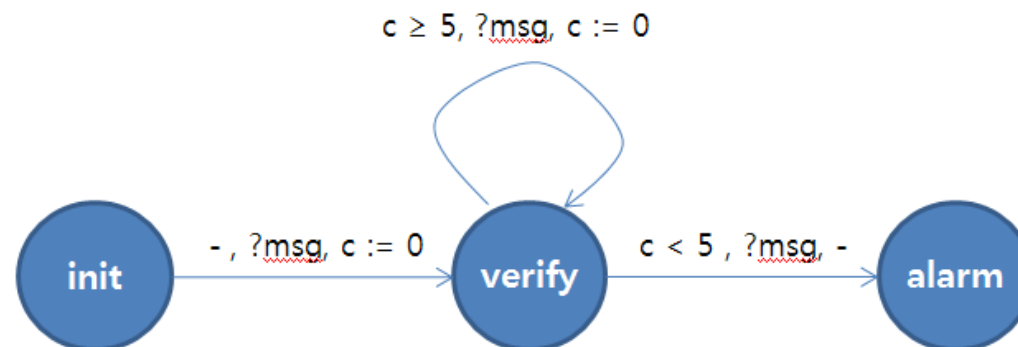
- Variables having non-negative real values in R
 - All clocks are null in the initial system states.
 - All clocks evolve at the same speed, synchronously with time.

- **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)



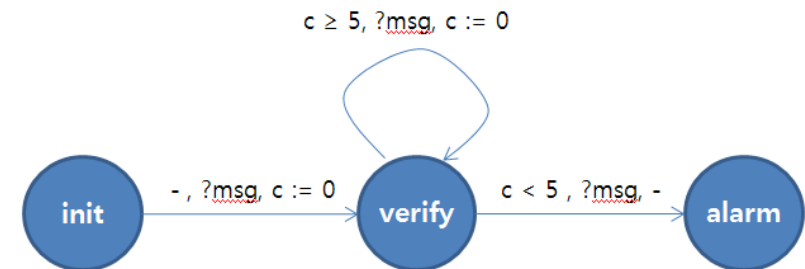
- Configurations and executions

- 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

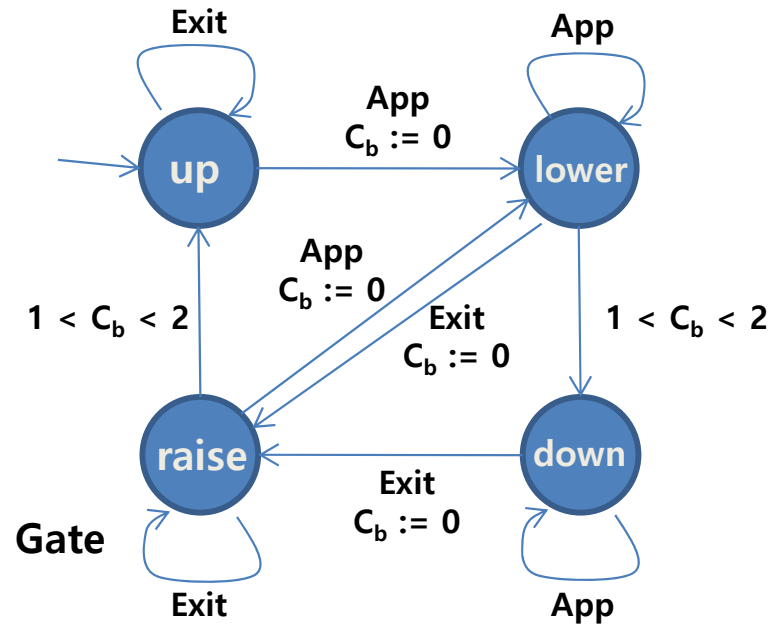
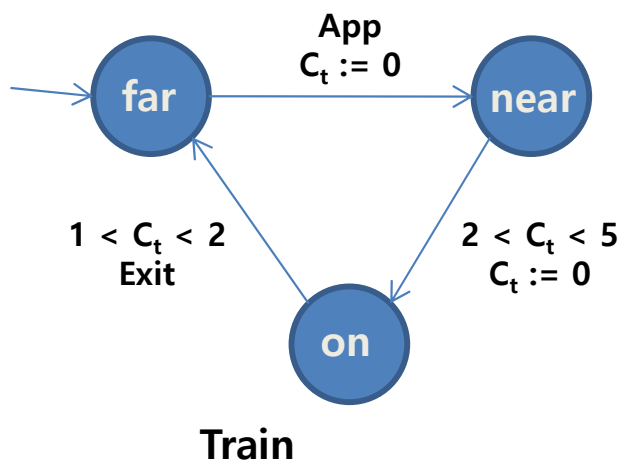
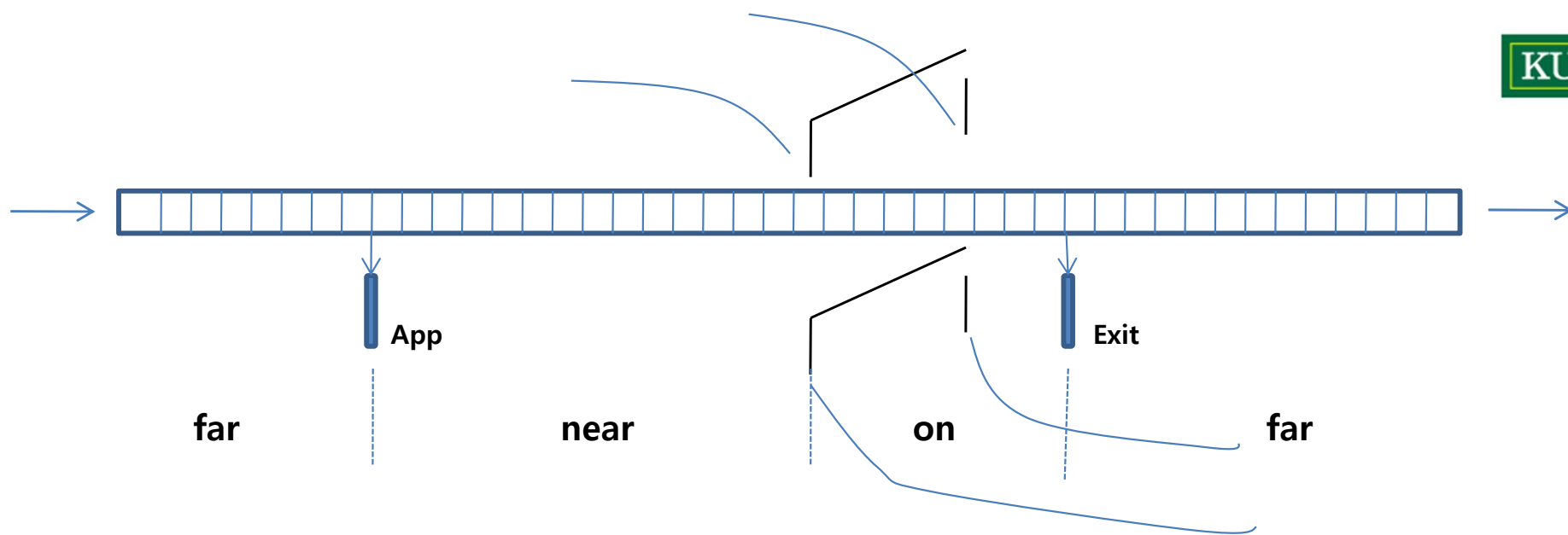
$(init, 0) \xrightarrow{\text{Delay transition}} (init, 10.2) \xrightarrow{?msg} (verify, 0) \xrightarrow{\text{Discrete transition}} (verify, 5.8) \xrightarrow{?msg} (verify, 0) \xrightarrow{\text{Discrete transition}} (verify, 3.1) \xrightarrow{?msg} (alarm, 3.1) \rightarrow \dots$

Delay transition

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

5.2 Networks of Timed Automata and Synchronization

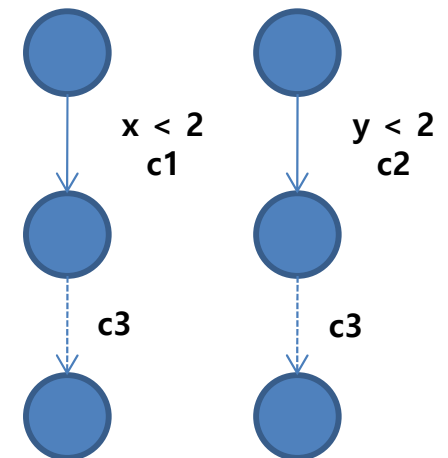
- It is useful to build a timed model in a composite fashion, by combining several parallel automata synchronized with one another.
→ **a timed automata network**
- Executions of a timed automata network
 - All automata components run in parallel at the same speed
 - Their clocks are all synchronized 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
- 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: $Ht < 5$), on (invariant: $Ht < 2$), lower/raise (invariant: $Hb < 2$)
- 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
- **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.”

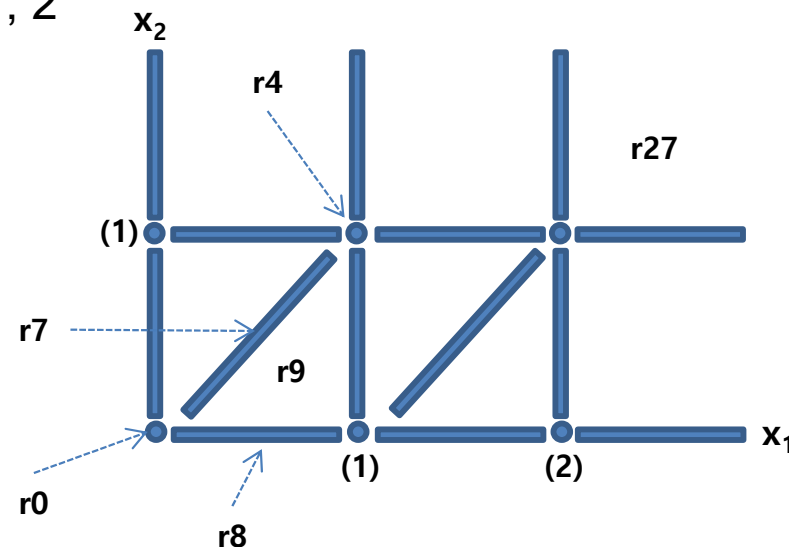
- Three ways to formally state real-time properties
 1. Express it in terms of the reachability of some sets of configurations
 2. Use observer automata in PLTL model checking
 - Given a property ϕ , a network R
 - Testing reachability of some states in the product $R \parallel A\phi$
 - UPPAAL, HYTECH
 3. Use a timed logic
 - TCTL (Timed CTL), etc.

- **TCTL (Timed CTL)**

- $\Phi, \Psi ::= P_1 \mid P_2 \mid \dots$ (atomic proposition)
 $\mid \neg\Phi \mid \Phi \wedge \Psi \mid \Phi \Rightarrow \Psi \mid \dots$ (boolean combinators)
 $\mid EF_{(\sim k)}\Phi \mid EG_{(\sim k)}\Phi \mid E\Phi U_{(\sim k)}\Psi$ (temporal combinators)
 $\mid AF_{(\sim k)}\Phi \mid AG_{(\sim k)}\Phi \mid A\Phi U_{(\sim k)}\Psi$ (path quantifiers)
- \sim : any comparison symbol from $\{<, \leq, =, \geq, >\}$
- k : any rational number from Q . (real number)
- Operator X does not exist in TCTL
- Example :
 - $AG(pb \Rightarrow AG_{(\leq 5)} alarm)$
 - "If a problem occurs, then the alarm will sound immediately and it will sound for at least 5 time units."
 - $AG(\neg far \Rightarrow AF_{(< 7)} far)$
 - "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
 - Clock values are unbounded
 - The set of real numbers used in clocks is dense
 - Overcome it with the equivalence classes, called “regions”
 - Example: $x_1, x_2 \sim k$ 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
 - UPPAAL
 - HyTech
 - (KRONOS)
 - SpaceEx (PHAver) ← for Hybrid System (CPS)

FORMAL VERIFICATION TOOLS AND CASE STUDIES

7. Formal Verification Tools

Introduction

- **Formal modeling methods and tools** (graphical methods only)

- SCR
- NuSCR (NuSRS)
- Statecharts Statemate MAGNUM)
- RSML / SpecTRM
- Petri-Nets (Design/CPN)
- Timed Automata (UPPAAL)

- **Formal verification** (model checking) **tools**

- SMV
- SPIN
- VIS
- CBMC
- DESIGN/CPN
- UPPAAL
- HyTech

SCR

■ Software Cost Reduction (SCR) ¹⁾

¹⁾ <http://www.nrl.navy.mil/itd/chacs/5546/SCR>

The screenshot shows the CHACS website interface. At the top, there is a navigation bar with 'CHACS Home', 'Sections', and 'Publications'. Below this is a breadcrumb trail: '/ NRL / ITD / CHACS / Sections / Software Engineering (5546) / Software Cost Reduction (SCR) Toolset'. The main heading is 'Software Cost Reduction (SCR) Toolset'. There are three tabs: 'Overview' (selected), 'System Requirements', and 'Component Overview'. The 'Overview' tab contains two paragraphs of text and a small icon. The first paragraph states that the toolset was created to help developers build specifications with greater assurance. The second paragraph describes the toolset as an integrated suite of tools for specifying and analyzing software requirements. Below the text is a 'Selected Publications' section with a table of three entries.

Center for High Assurance Computer Systems

CHACS Home | Sections | Publications

/ NRL / ITD / CHACS / Sections / Software Engineering (5546) / Software Cost Reduction (SCR) Toolset

Software Cost Reduction (SCR) Toolset

Overview | System Requirements | Component Overview

This toolset was created to help developers build specifications with greater assurance that the requirements will be complete, and the resulting software error-free. This is especially important for critical systems. The method for creating specifications is based on a scalable tabular notation.

The toolset is an integrated suite of tools for specifying and analyzing software requirements. The table-based system uses a language that engineers and computer system developers can quickly feel comfortable with. While the algorithms used in the toolset are based in mathematical logic, the user does not need to know this to successfully use the toolset.

The toolset consists of an editor for creating specifications, a consistency and completeness checker, a browser for picturing dependencies, and a simulator for testing the created specification. Some versions also include model checking and theorem proving extensions.

Selected Publications

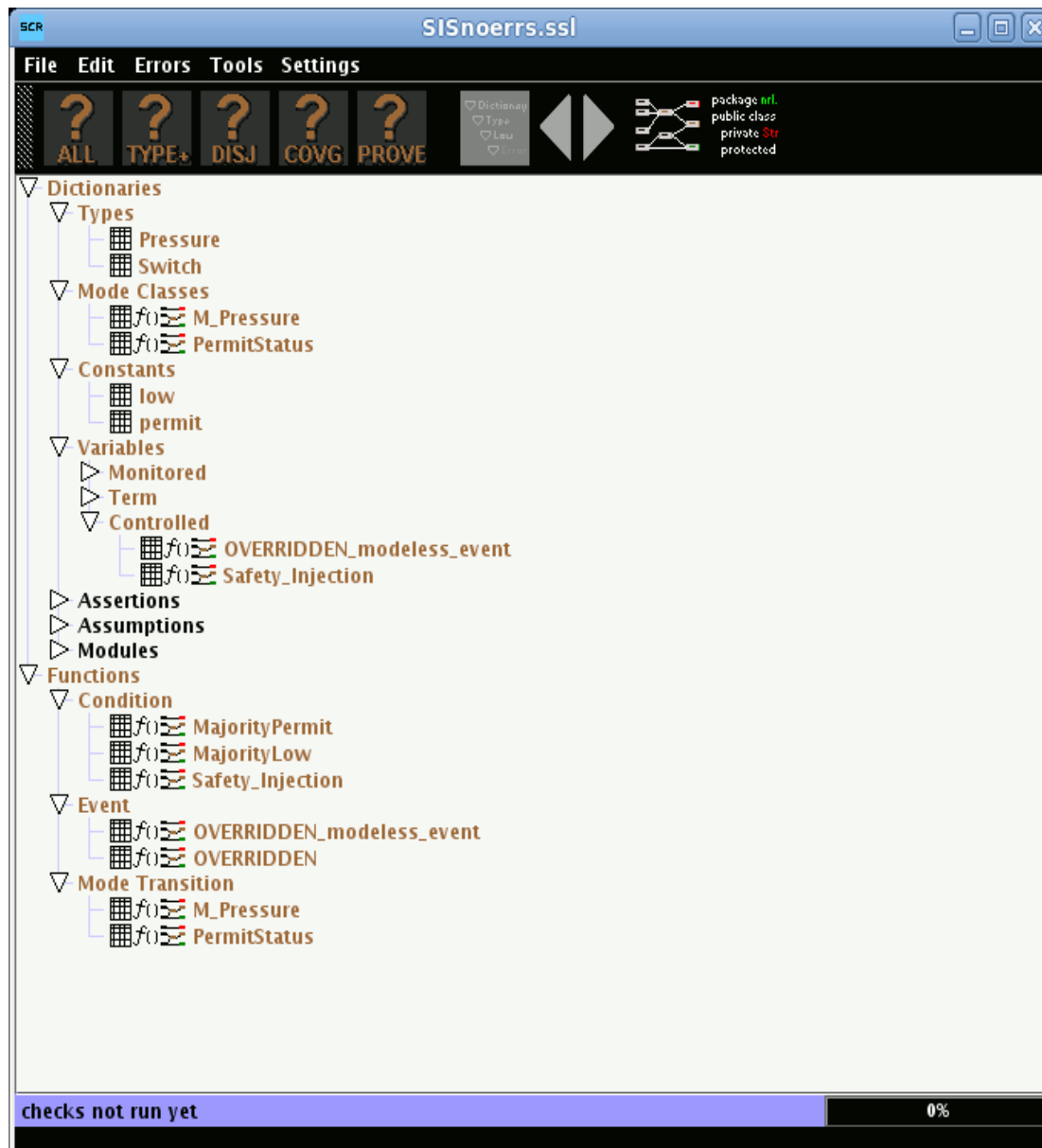
2007	Heitmeyer CL, Jeffords R. 2007. Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned. 2007 IEEE Aerospace Conference.
2005	Heitmeyer CL, Archer M, Bharadwaj R, Jeffords R. 2005. Tools for constructing requirements specifications: The SCR toolset at the age of ten. International Journal of Computer Systems Science and Engineering. 20(1):19-35.
2002	Heitmeyer CL, Marciniak JJ. 2002. Software Cost Reduction. Encyclopedia of Software Engineering. 2

CHACS Home
Contact Us
Mission Statement
Objectives
Demographics
Focus Areas
Major Accomplishments

Sections

- COMSEC Systems (5541)
- Computer Security (5542)
- Formal Methods (5543)
- Network Security (5544)
- Mobile Systems Security (5545)
- Software Engineering (5546)
 - Software Cost Reduction (SCR) Toolset**
 - System Requirements
 - Component Overview

Publications



- Main Window**
- Condition table
 - Event table
 - Mode Transition table

Editor

The specification consists of two basic types of tables. The first is dictionaries, which describe the types used in the specification, variables, and other objects used to describe the specification with the second table type.

The second type of table describes functions; that is, given certain events or conditions, and the current state of the application, what is the output value of a given variable?

The tables include editing functions for building and changing the developing specification.

The screenshot shows a software window titled "Variable Dictionary" with a menu bar containing "File". Below the menu bar is a toolbar with a checkmark icon, a tree icon, and a "TYPE+" label. To the right of the toolbar are three checkboxes: "Mon", "Term", and "Con", and a "Match Case" checkbox. Below these are two dropdown menus labeled "Name" and "Contains".

Name	DGB	Class	Type	Initial Value	Accuracy	Table	Kind	Value	Comment
Block	DGB	Mon	Switch	OFF	N/A	-			
MajorityLow	DGB	Term	Boolean	TRUE	0	T	Table		
MajorityPermit	DGB	Term	Boolean	TRUE	0	T	Table		
OVERRIDDEN	DGB	Term	Boolean	TRUE	0	T	Table		
OVERRIDDEN_model ess_event	DGB	Con	Boolean	TRUE	0	T	Table		
PB	DGB	Mon	Pressure	14	0.05%	-			
PG	DGB	Mon	Pressure	14	0.05%	-			
PR	DGB	Mon	Pressure	14	0.05%	-			
Reset	DGB	Mon	Switch	OFF	N/A	-			
Safety_Injection	DGB	Con	Switch	OFF	N/A	T	Table		

Below the table is a "Notes" section with a large empty text area.

Consistency Checker

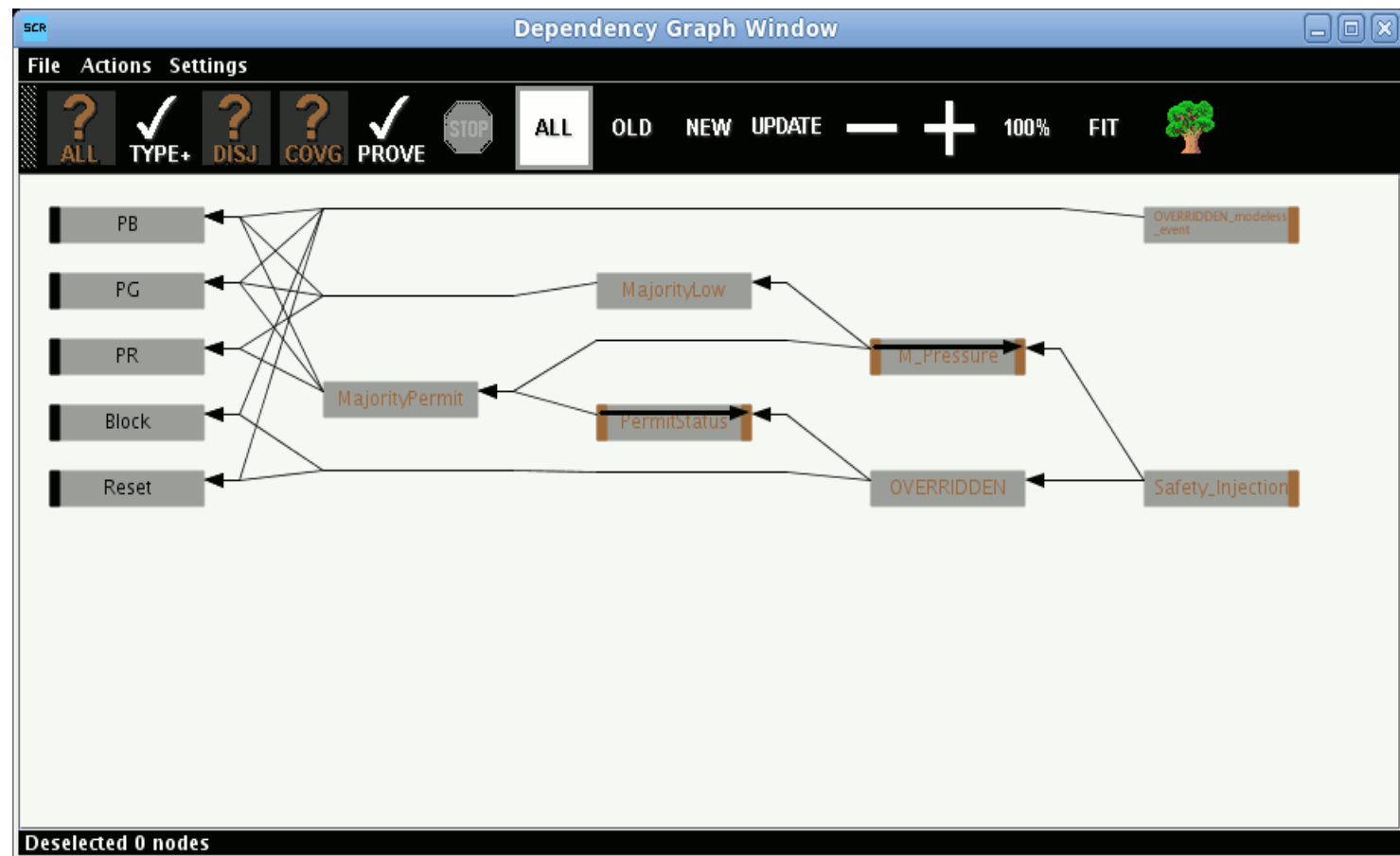
This checks for correctness and consistency in the tables that describe the specification. Kinds of checks done are syntax and type checking, circular definitions, missing cases, undefined variables, and non-determinism. Any errors found are displayed in the main window under the affected object, making it easy for the user to locate and correct the error.

Dependency Browser

An existing specification is analyzed to find which objects in the specification depend on which other objects (eg, variables), and a graph of the dependency relationship is created and displayed by this component.

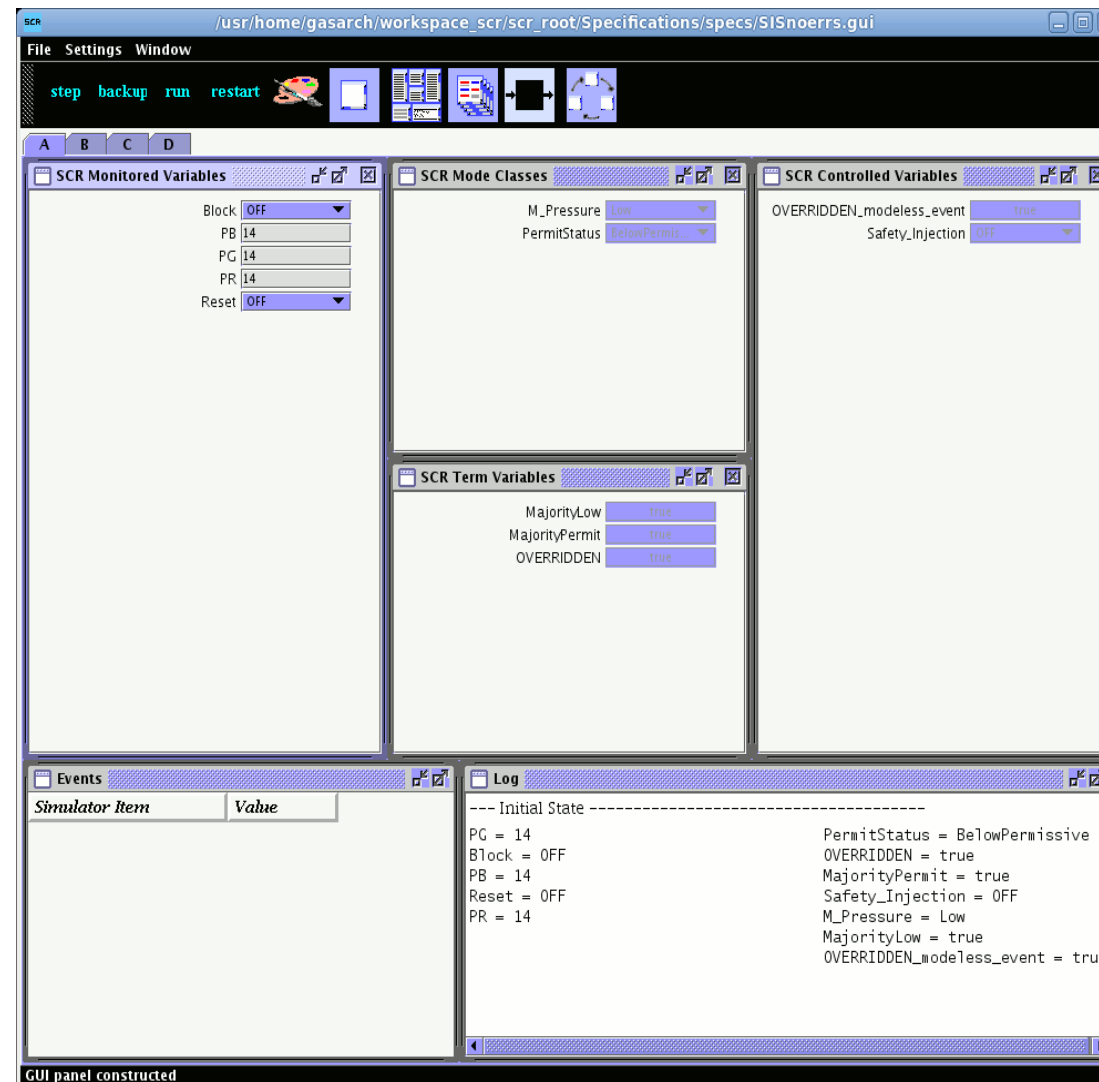
This allows the user to create scenarios, and play them out to check that the specification behaves as expected. The simulator can be run from within the SCR toolset, or it can run separately when code is generated from the specification (one of the functions of the toolset).

Simulator



Simulator

The simulator display takes two basic forms. One is an automatically generated tabular display, [associate with the simulator.gif] and the other is graphical, tailored to specific applications. This version requires user extensions, based on a framework built into the toolset.



NuSCR

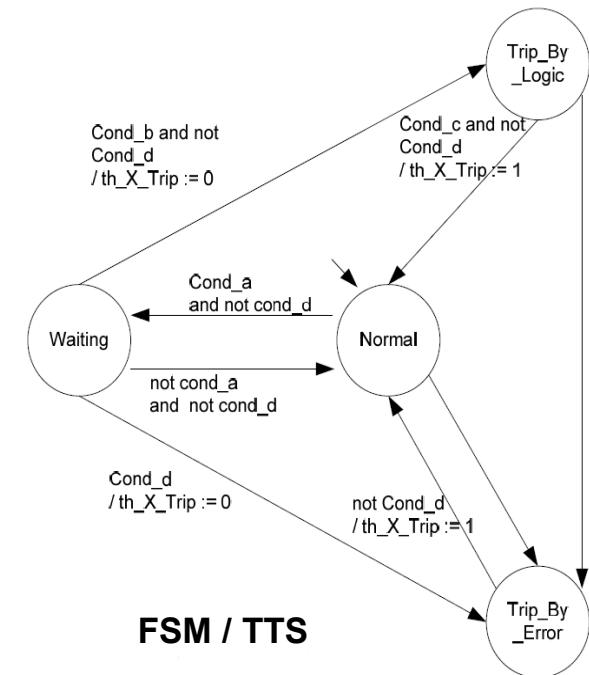
■ NuSCR

A Formal requirements specification language customized for nuclear I&C systems

- Customizing SCR for nuclear domain
- Consisting of 4 constructs
 - + SDT (Structured Decision Table)
 - + FSM (Finite State Machine)
 - + TTS (Timed Transition System)
 - + FOD (Function Overview Diagram)
- Various Supporting for seamless verification and safety analysis
- A starting point of the NuDE framework

Conditions		
$k_X_MIN \leq f_X \leq k_X_MAX$	T	F
Actions		
$f_X_Valid := 0$	X	
$f_X_Valid := 1$		X

SDT



FSM / TTS

NuSRS - Eclipse Platform

File Edit Navigate Search Project NuDE NuSRS Run Window Help

Quick Access Resource FBD_Editor NuDE 0.9 NuSRS

Common Navigator

- New_Nude
 - FBDtoC
 - FBDtoVerilog
 - NuSCRtoFBD
 - NuSRS
 - (NuSCR) RPS BP (20130716).xml

Hierarchy Window

- Root
 - g_BP
 - g_LO_SG1_LEVEL
 - g_VAR_OVER_PWR
 - g_HI_LOG_POWER
 - g_LO_PZR_PRESS
 - g_SG1_LO_FLOW
 - g_HI_LOCAL_POWER

Description Window

- g_VAR_OVER_PWR
 - Description
 - 가변 과출력 트립 (자동비율/상승)
 - TemplateNumber
 - Input
 - f_VAR_OVER_PWR_PV : 0..30
 - f_VAR_OVER_PWR_Manu_Te
 - f_VAR_OVER_PWR_MT_Query
 - f_VAR_OVER_PWR_Trip_Stati
 - f_VAR_OVER_PWR_Ptrp_Stati
 - f_Mod_Err : boolean
 - f_VAR_OVER_PWR_Chan_Err
 - f_VAR_OVER_PWR_Op_Byp_I
 - Output
 - f_VAR_OVER_PWR_Val_Out :
 - f_VAR_OVER_PWR_Ptrp_SP :
 - f_VAR_OVER_PWR_Trip_SP :
 - th_VAR_OVER_PWR_Ptrp_Lo
 - th_VAR_OVER_PWR_Trip_Lo
 - f_VAR_OVER_PWR_PV_Err : t
 - f_VAR_OVER_PWR_Trip_Out :

Diagram Window

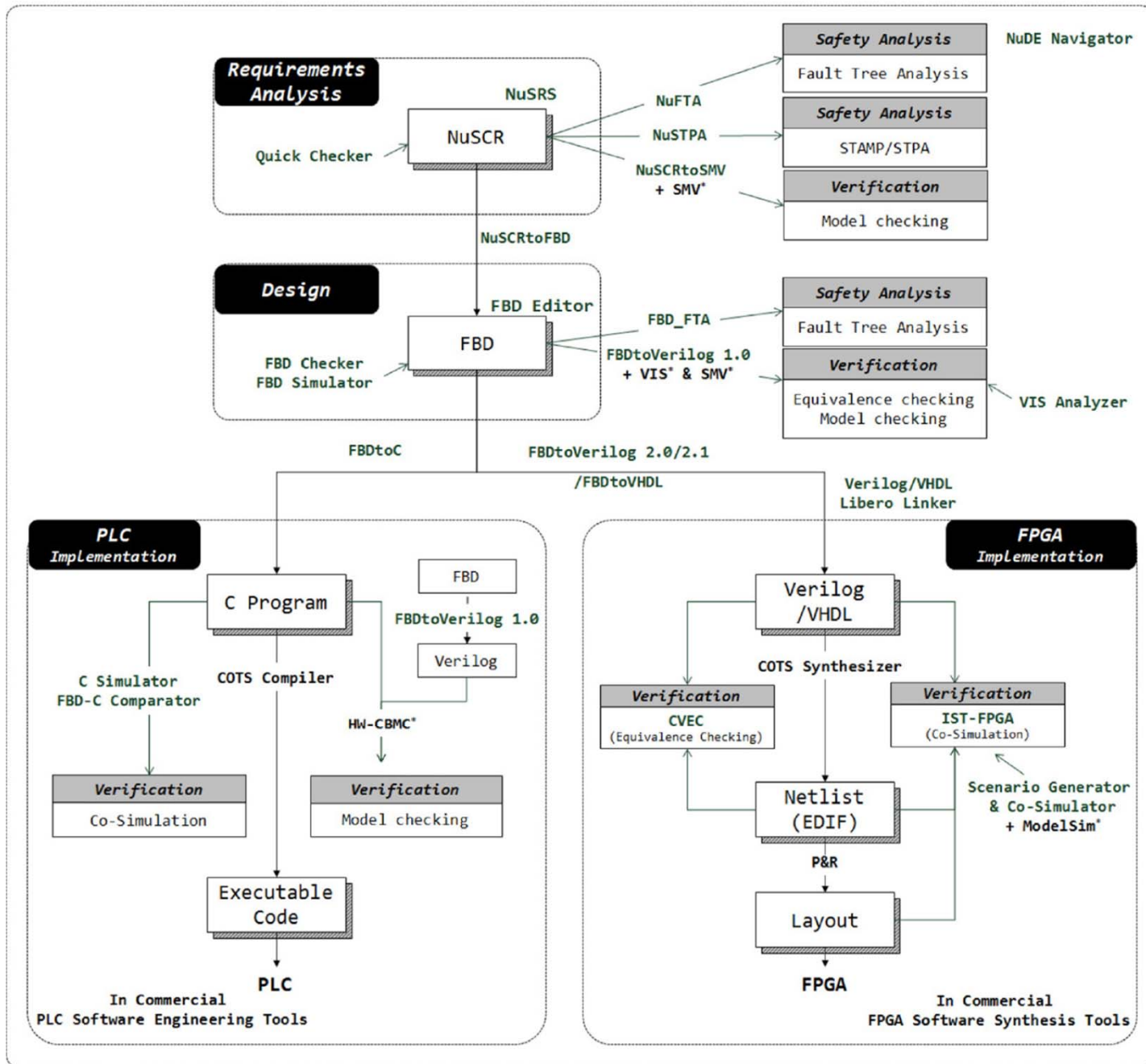
Type Window

```

f_Mod_Err : boolean
f_VAR_OVER_PWR_Chan_Err : boolean
f_VAR_OVER_PWR_MT_Query : boolean
f_VAR_OVER_PWR_Manu_Test : 0..30000
f_VAR_OVER_PWR_Op_Byp_Init : boolean
f_VAR_OVER_PWR_PV : 0..30000
f_VAR_OVER_PWR_PV_Err : boolean
f_VAR_OVER_PWR_Ptrp_Out : boolean
f_VAR_OVER_PWR_Ptrp_SP : 0..30000
f_VAR_OVER_PWR_Ptrp_Status : boolean
f_VAR_OVER_PWR_Trip_Out : boolean
f_VAR_OVER_PWR_Trip_SP : 0..30000
f_VAR_OVER_PWR_Trip_Status : boolean
  
```

0 items selected

NuSRS – NuSCR Modeling Environment

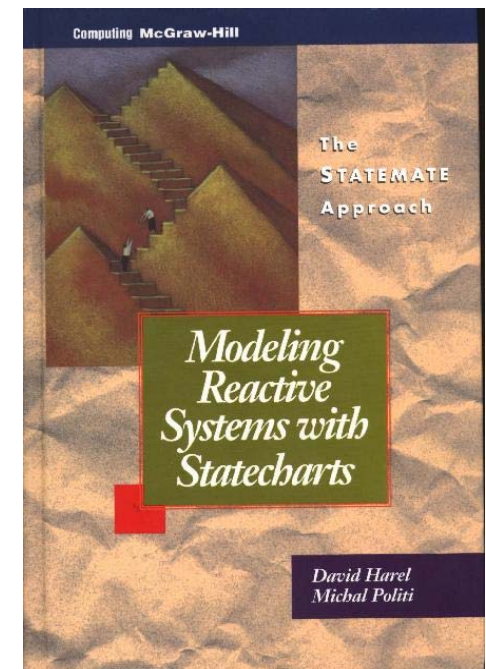


Statecharts

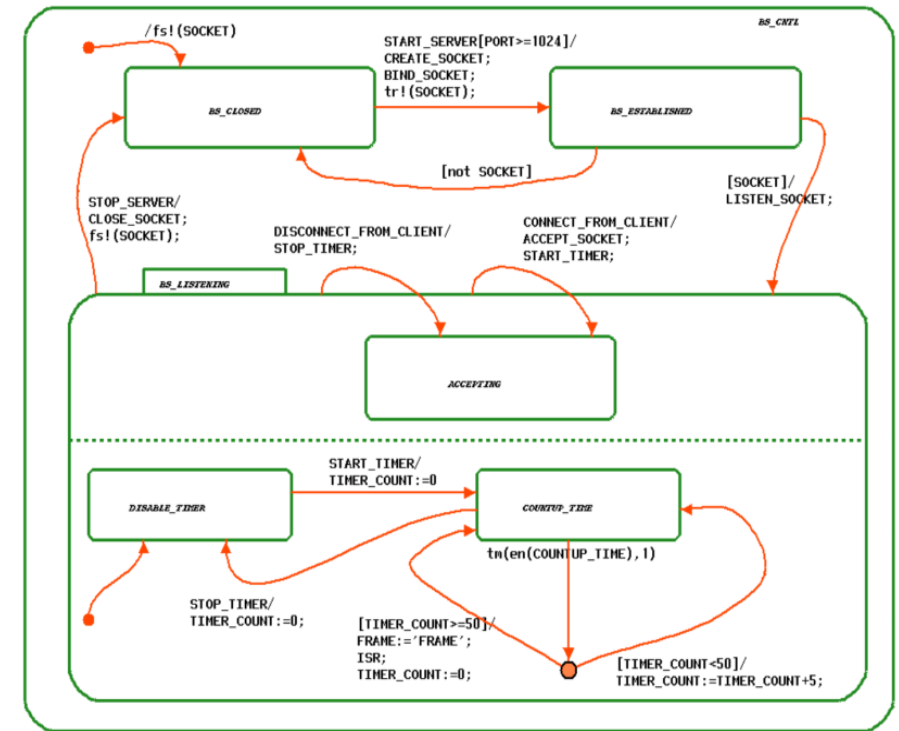
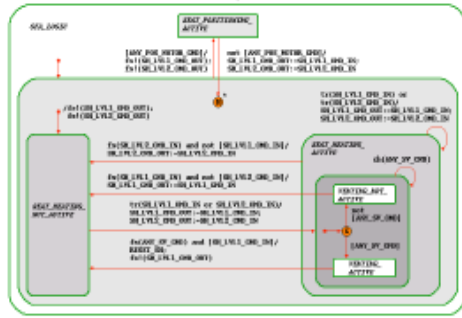
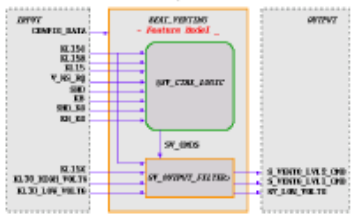
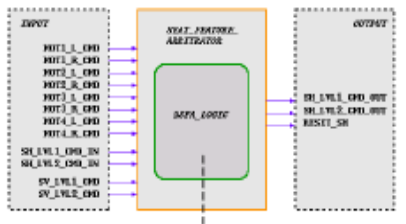
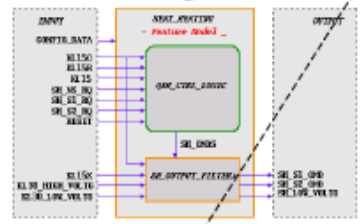
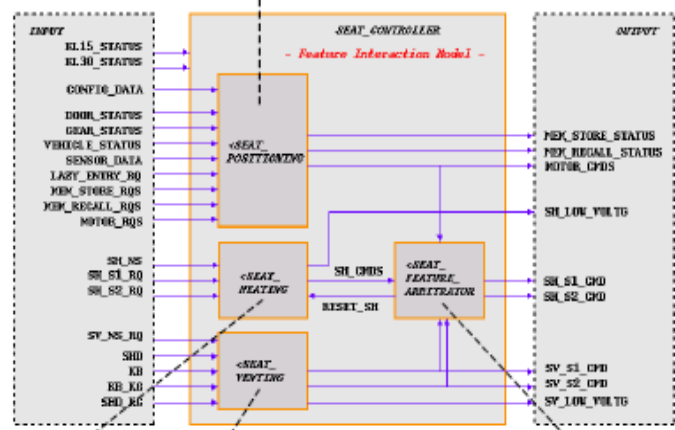
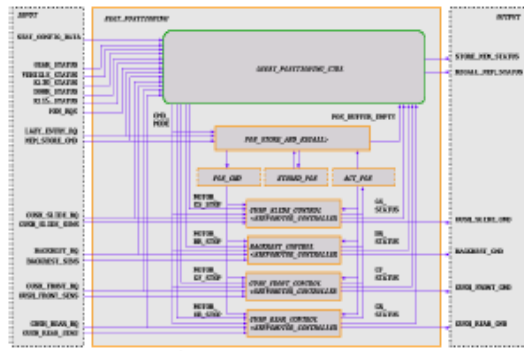
■ IBM Rational Statemate MAGNUM

A graphical working environment developed by David Harel

- Create a visual, graphical specification that clearly and precisely represents the intended functions and behavior of the system
- The Statecharts specification may be executed, or graphically simulated
- The 3 views of the system mode
 - + Module-charts
 - + Activity-charts
 - + Statecharts
- Generates C, Ada, VHDL and Verilog code
- Formal verification through in-house model checker



Activity-chart



Statechart

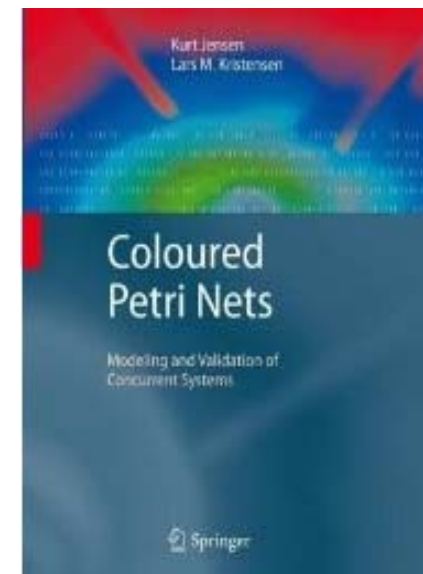
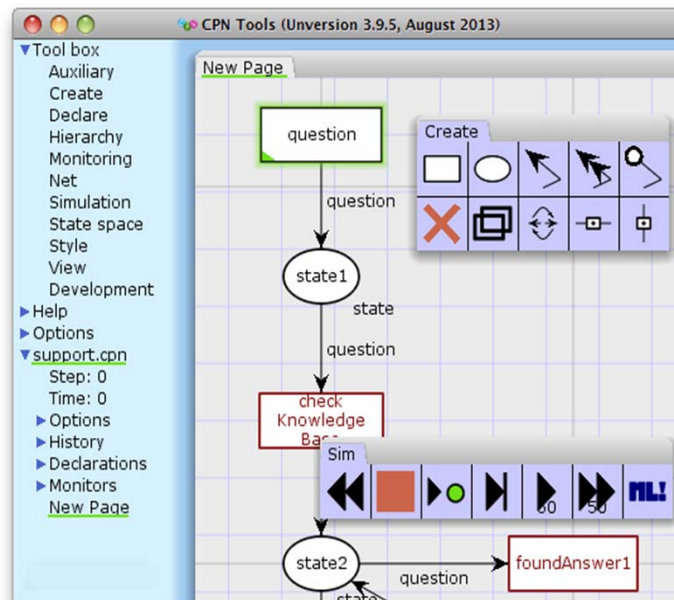
Design/CPN

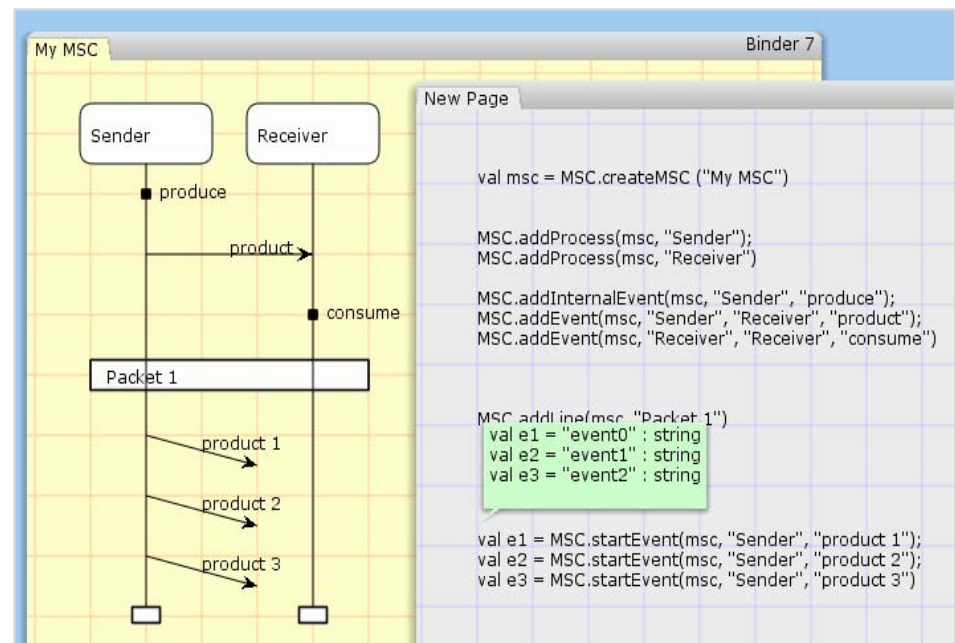
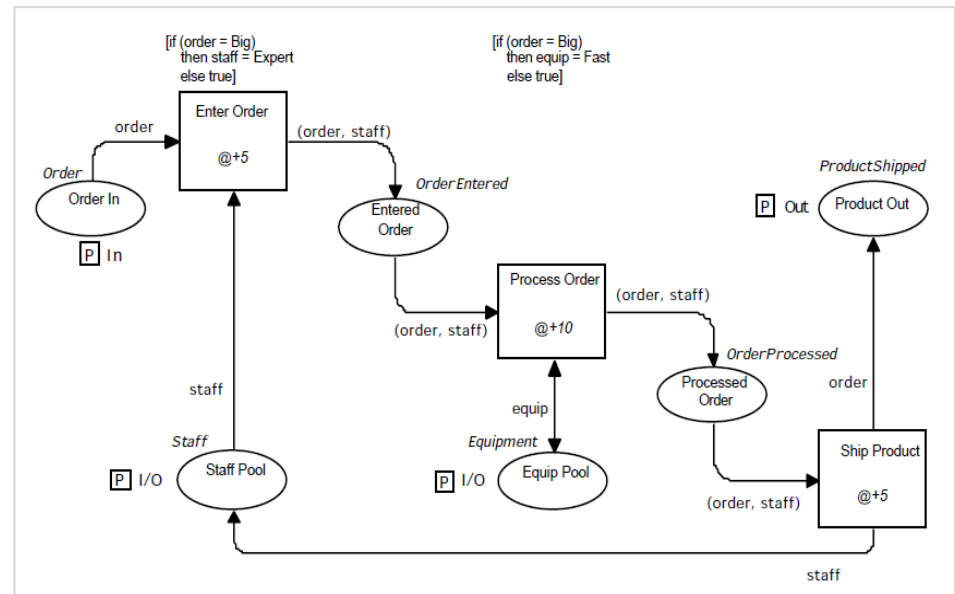
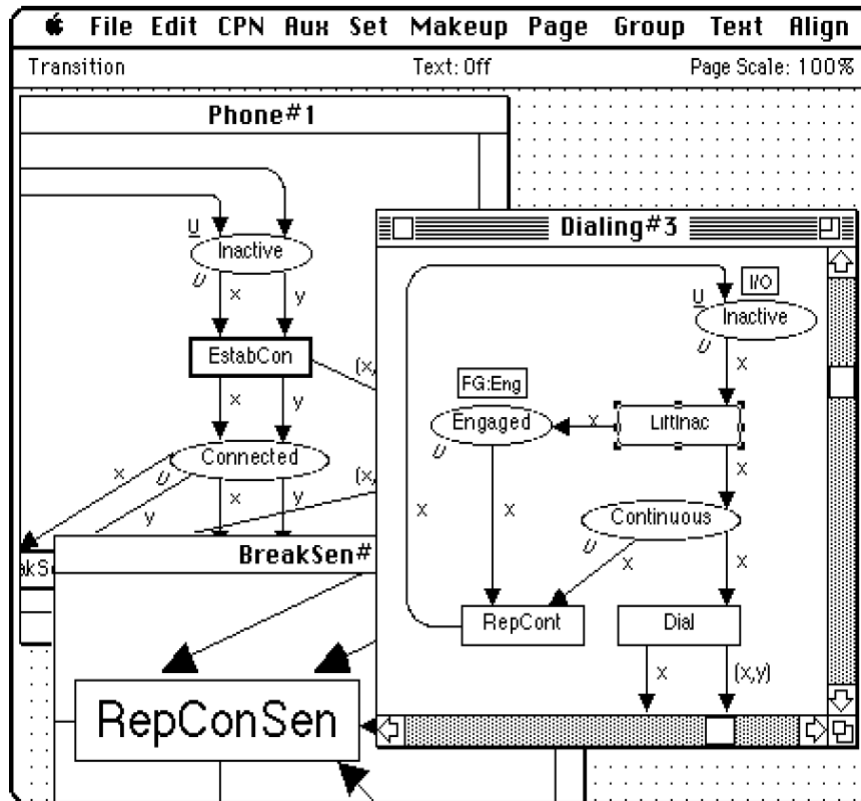
■ Design/CPN (CPN Tools 4.0)

¹⁾ <http://cpntools.org/>

A tool for editing, simulating, and analyzing Colored Petri nets¹⁾

- CPN Editor : construction, modification and syntax check of CPN models (CPN Editor)
- CPN Simulator : interactive and automatic simulation of CPN models (CPN Simulator)
+ simulation-based performance analysis of CPN models
- Occurrence Graph Tool : construction and analysis of occurrence graphs for CPN models
→ state spaces or reachability graphs/trees





Timed Automata

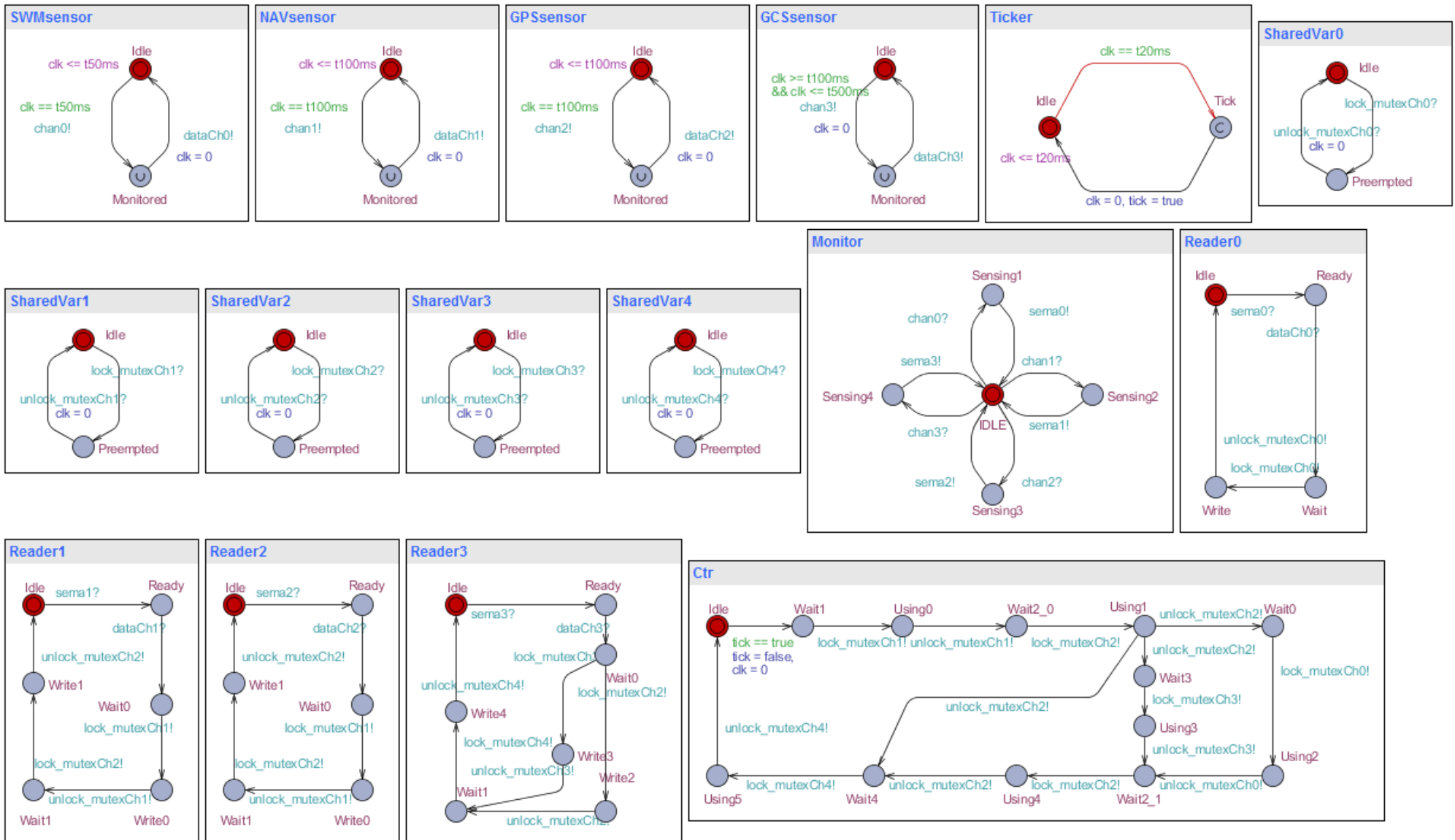
■ Timed Automata (UPPAAL)

¹⁾ <http://www.uppaal.org/>

An integrated tool environment for modeling, validation and verification of **real-time systems** modeled as networks of **timed automata**, extended with data types (bounded integers, arrays, etc.) ¹⁾

- Use timed automata to analyze timed systems
- Graphical editor
- Graphical simulator
- Verifier
- Model checking
 - + CTL reachability analysis based on AG / EF





Timed automata models in UPPAAL

D:\ITRC\건국대 ITRC\uppaal/new/013.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Overview

```

A[] !(Ctr.Wait4 && Ctr.clk > t20ms && Reader2.Write4)
A[] !(Ctr.Wait3 && Ctr.clk > t20ms && Reader2.Write3)
A[] !((Ctr.Wait2_0 || Ctr.Wait2_1) && Ctr.clk > t20ms && (Reader0.Write1 || ...
A[] !(Ctr.Wait1 && Ctr.clk > t20ms && (Reader0.Write1 || Reader1.Write1))
A[] !(Ctr.Wait0 && Ctr.clk > t20ms && Reader3.Write)

```

Check
Insert
Remove
Comments

Query

```
A[] !(Ctr.Wait4 && Ctr.clk > t20ms && Reader2.Write4)
```

Comment

```
Controller doesn't wait to access SharedVar4 over 20 ms while Reader2 accesses SharedVar4
```

Status

```

A[] !(Ctr.Wait4 && Ctr.clk > t20ms && Reader2.Write4)
Property is not satisfied,
A[] !(Ctr.Wait3 && Ctr.clk > t20ms && Reader2.Write3)
Property is not satisfied,
A[] !((Ctr.Wait2_0 || Ctr.Wait2_1) && Ctr.clk > t20ms && (Reader0.Write1 || Reader1.Write1 || Reader2.Write2)
Property is not satisfied,
A[] !(Ctr.Wait1 && Ctr.clk > t20ms && (Reader0.Write1 || Reader1.Write1))
Property is not satisfied,
A[] !(Ctr.Wait0 && Ctr.clk > t20ms && Reader3.Write)
Property is not satisfied,

```

D:/ITRC/건국대 ITRC/uppaal/new/013.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

Ticker
chan0: NAVsensor --> Monitor
chan1: GPSsensor --> Monitor
chan3: SWMsensor --> Monitor
unlock_mutexCh3: Reader2 --> SharedVar3

Next Reset

Simulation Trace

```

(tick, tick, tick, tick, tick, Preempted, tick, tick,
unlock_mutexCh1: Reader2 --> SharedVar1
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,
lock_mutexCh3: Reader2 --> SharedVar3
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,
unlock_mutexCh4: Reader3 --> SharedVar4
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,
lock_mutexCh4: Ctr --> SharedVar4
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,
lock_mutexCh1: Ctr --> SharedVar1
(Idle, Idle, Idle, Idle, Idle, Preempted, Idle, Preempted,
unlock_mutexCh1: Ctr --> SharedVar1
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,
unlock_mutexCh1: Ctr --> SharedVar1
(Idle, Idle, Idle, Idle, Idle, Idle, Idle, Preempted,

```

Trace File:

Prev Next Replay
Open Save Auto

Slow Fast

Drag out

```

tick = 1
NAVsensor, interval = 100
GPSsensor, interval = 100
GCSsensor, minInterval =
GCSsensor, maxInterval =
SWMsensor, interval = 50
Ticker, interval = 20
Ctr, interval = 20
NAVsensor, clk = 100
GPSsensor, clk = 100
GCSsensor, clk = 0
SWMsensor, clk = 50
SharedVar0, clk in [20,40]
SharedVar1, clk = 0
SharedVar2, clk = 100
SharedVar3, clk in [50,60]
SharedVar4, clk = 0
Ticker, clk = 20
Ctr, clk in [20,40]
NAVsensor, clk = GPSsensor,
GPSsensor, clk - GCSsensor,
SWMsensor, clk - GCSsensor,
SWMsensor, clk - SharedVar0,
SharedVar0, clk - Ctr, clk in
SharedVar2, clk - SharedVar2,
SharedVar2, clk - SharedVar4,
Ticker, clk - SharedVar4, clk

```

NAVsensor

GPSsensor

GCSsensor

SWMsensor

SharedVar0

SharedVar1

SharedVar2

SharedVar3

SharedVar4

Monitor

Reader0

Reader1

Reader2

Reader3

Ticker

Ctr

SharedVar0 SharedVar1 SharedVar2 SharedVar3 SharedVar4 Monitor Reader0 Reader1 Reader2

Idle Idle Idle Preempted Preempted IDLE Idle Idle Write4

lock_mutexCh3

unlock_mutexCh3


SMV

■ SMV

¹⁾ <http://www.cs.cmu.edu/~modelcheck/smv.html>

A symbolic model checker of CLT formulae on networks of automata with shared variables¹⁾

- The first model checker using the BDD technology
- Suited for fully checking a complex system
- Input : **SMV input program language** or **Verilog program**
- CTL model checking
- No support for (systematic) simulating

Carnegie Mellon
Model Checking @CMU


[\[Home\]](#)
[\[People\]](#)
[\[Software\]](#)
[\[Publications\]](#)
[\[Support\]](#)
[\[Links\]](#)
[\[Internal\]](#)

The SMV System

```

File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups

Name Layer
├─ (top_level)
│ └─ cycle
│   └─ f_Mod_Err
│     └─ f_VAR_OVER_PWR_Chan_Err
│       └─ f_VAR_OVER_PWR_MT_Query
│         └─ f_VAR_OVER_PWR_Manu_Test
└─

Source Trace Log

File Show
-- SMV Input for g_VAR_OVER_PWR
-- SMV Input for f_VAR_OVER_PWR_Val_Out

MODULE m_f_VAR_OVER_PWR_Val_Out(f_VAR_OVER_PWR_PV, f_VAR_OVER_PWR_Manu_Test, f_VAR_OVER_PWR_MT_Query, cycle, sec)
VAR
  f_VAR_OVER_PWR_Val_Out : 0..100;
  -- inputs

STATE : (_init_, s0, s1);

ASSIGN

init(STATE) := _init_;
next(STATE) := case
  FROM-_init_-TO-s0-taken : s0;
  FROM-s0-TO-s0-taken : s0;
  FROM-s1-TO-s0-taken : s0;
  FROM-_init_-TO-s1-taken : s1;
  FROM-s0-TO-s1-taken : s1;
  FROM-s1-TO-s1-taken : s1;
  1 : STATE;
esac;

-- Outputs
init(f_VAR_OVER_PWR_Val_Out) := 0;
  
```

The SMV input program (Cadence SMV)

All results

Property	Result	Time
(EF ((state=operate1)&(alarm=1)))	false	Sat Dec 10 20:30:25 64비트 시스템 1월 11일 2011
(EF ((state=connect1)&(alarm=1)))	true	Sat Dec 10 20:30:25 64비트 시스템 1월 11일 2011
(AF ((temp<65)&!(state=create)))	false	Sat Dec 10 20:30:25 64비트 시스템 1월 11일 2011

File	Edit	Run	View																	
	: 1	2	3	4	5 :															
Loc.x	0	0	0	0	0															
Loc.y	0	0	0	0	0															
ack2	-	-	-	-	-															
alarm	0	0	0	0	0															
obs	0	64	0	0	0															
state	ready	notice	create	operate2	connect2															
temp	68	68	0	68	68															

i-search:

The result (counterexample) of the CTL model checking

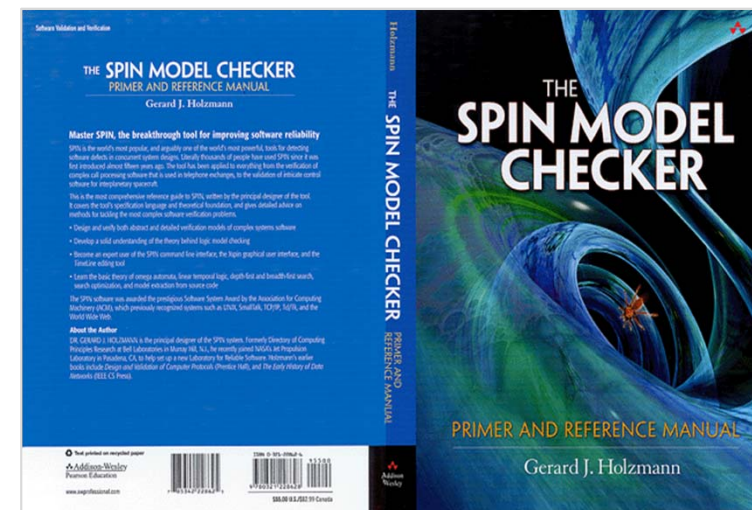
SPIN

■ SPIN

¹⁾ <http://spinroot.com/spin/whatispin.html>

A tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols ¹⁾

- Developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980.
- The system is described in a modeling language called **Promela** (Process Meta Language) + Allowing for the dynamic creation of concurrent processes
- **Communication via message channels** can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).
- Simulator
- Verification: LTL model checking or assertion




```

SPIN CONTROL 5.2.3 -- 25 November 2009
File.. Edit.. View.. Run.. Help SPIN DESIGN VERIFICATION
^
  assert( (mutex_0 != 2) &&
          (mutex_1 != 2) &&
          (mutex_2 != 2) &&
          (mutex_3 != 2) &&
          (mutex_4 != 2) )
}
init
{
  chan sema0 = [0] of {bit};
  chan sema1 = [0] of {bit};
  chan sema2 = [0] of {bit};
  chan sema3 = [0] of {bit};
  atomic{
    run monitor(sema0, sema1, sema2, sema3);

    run reader0(sema0);
    run reader1(sema1);
    run reader2(sema2);
    run reader3(sema3);
    run controller()
  }
}
+ <done preprocess>
- <stop simulation>
- <done>

```

The PROMELA program

Linear Time Temporal Logic Formulae

Formula: Load...

Operators:

Property holds for: All Executions (desired behavior) No Executions (error behavior)

Notes:

Use Load to open a file or a template.

Symbol Definitions:

```

#define sensor_send sensor[0] == 1
#define reader_rcv semaphore0 == 1

```

Never Claim:

```

/*
 * Formula As Typed: [] (sensor_send -> <> reader_rcv)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([] (sensor_send -> <> reader_rcv))
 * (formalizing violations of the original)
 */
never { /* !([] (sensor_send -> <> reader_rcv)) */

```

Verification Result: valid

```

never claim          +
assertion violations + (if within scope of claim)
acceptance cycles   + (fairness disabled)
invalid end states  - (disabled by never claim)

```

State-vector 64 byte, depth reached 130883, errors: 0
800471 states, stored (848676 visited)

Simulation Output

Search for: Find

```

1089: proc 6 (reader_4) line 29 "pan_in" (state 5) [mutex_A = 1]
1089: proc 6 (reader_4) line 37 "pan_in" (state 8) [mutex_A = 0]
1090: proc 6 (reader_4) line 111 "pan_in" (state 12) [sema = 0]
1091: proc 6 (reader_4) line 112 "pan_in" (state 14) [.(goto)]
1092: proc 2 (monitor) line 121 "pan_in" (state 73) [(1)]
1093: proc 2 (monitor) line 120 "pan_in" (state 68) [data = 3]
1094: proc 5 (reader_3) line 102 "pan_in" (state 32) [sema = 0]
1095: proc 5 (reader_3) line 103 "pan_in" (state 34) [.(goto)]
1096: proc 2 (monitor) line 121 "pan_in" (state 72) [.(goto)]
1097: proc 2 (monitor) line 138 "pan_in" (state 89) [.(goto)]
1098: proc 2 (monitor) line 131 "pan_in" (state 88) [((data==3))]
1099: proc 2 (monitor) line 134 "pan_in" (state -) [values: 3!1]
1099: proc 2 (monitor) line 134 "pan_in" (state 45) [sema_ch?sema]
1100: proc 5 (reader_3) line 98 "pan_in" (state -) [values: 3?1]
1100: proc 5 (reader_3) line 97 "pan_in" (state 33) [sema_ch?sema]
1101: proc 5 (reader_3) line 67 "pan_in" (state 11) [((mutex_D==0))]

```

Single Step Run Save in: sim.out Clear Cancel

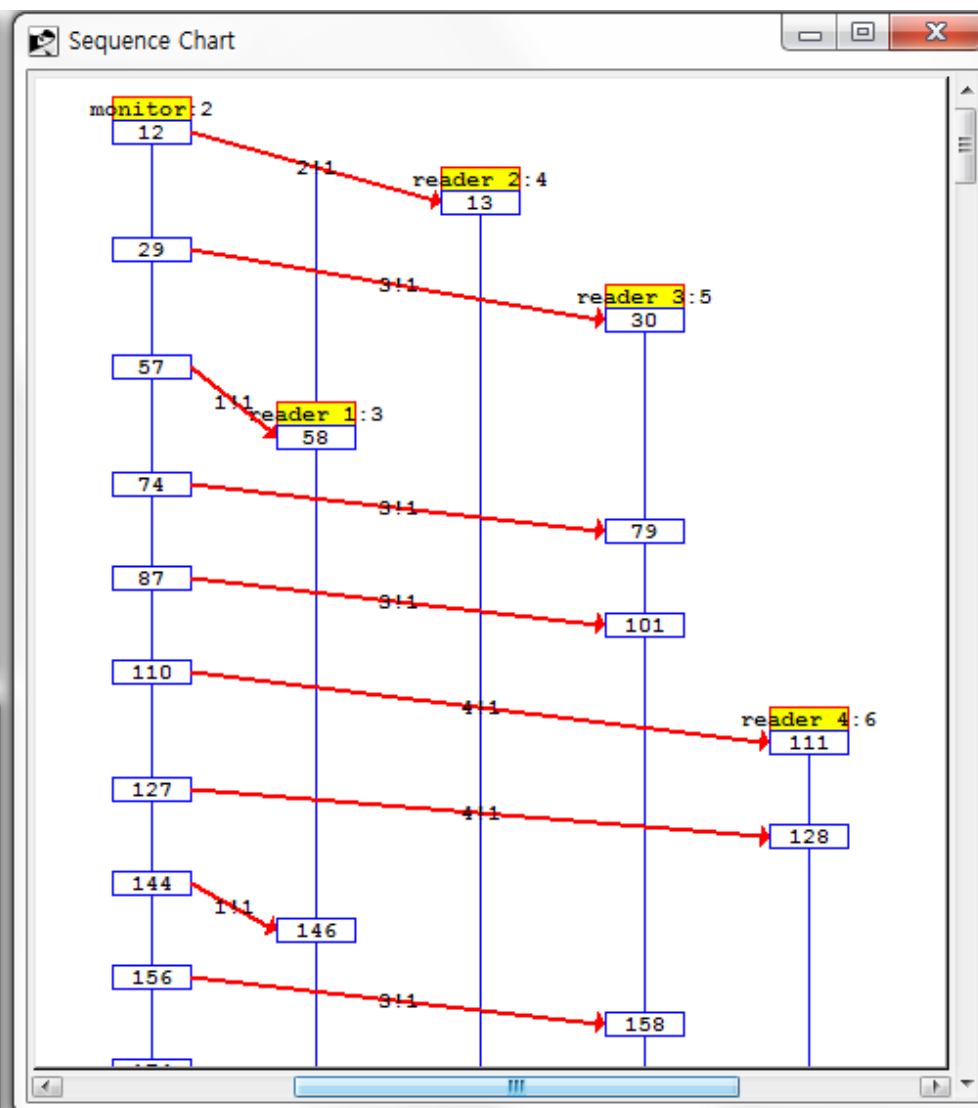
Data Values

Search for: Find

```

data = 3
mutex_A = 0
mutex_B = 0
mutex_C = 0
mutex_D = 0
mutex_E = 0

```



The verification/simulation results with sequence diagram

VIS

■ VIS

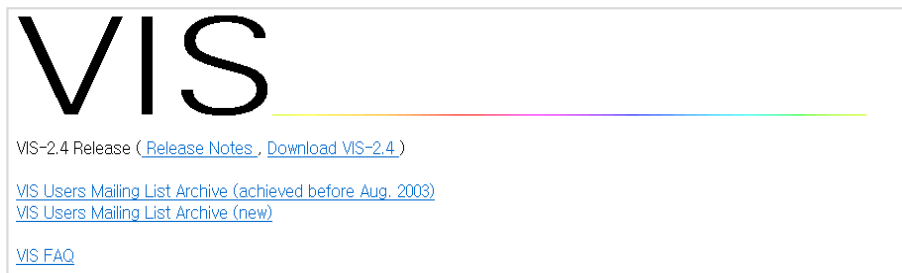
¹⁾ <http://vlsi.colorado.edu/~vis/>

A system for formal verification, synthesis, and simulation of finite state systems ¹⁾

- Simulation of logic circuits (proof of concept only)
- Formal "implementation" verification of combinational and sequential circuits (proof of concept only)
- State-of-the-art formal "design" verification using fair CTL model checking and language emptiness
- Logic synthesis via hierarchy restructuring and a path to and from SIS

- Input : **Verilog** HDL through vl2mv (into **BLIF-MV** format)

- No GUI



VIS

VIS-2.4 Release ([Release Notes](#) , [Download VIS-2.4](#))

[VIS Users Mailing List Archive \(achieved before Aug. 2003\)](#)
[VIS Users Mailing List Archive \(new\)](#)

[VIS FAQ](#)

```

vis release 2.0 (compiled Thu Jun 26 11:08:16 2008)
vis> read_blif_mv h_X_Pretrip_Manual.mv
vis> flatten_hierarchy
vis> static_order
vis> build_partition_mdds
vis> simulate -i inputVector1.txt
# vis release 2.0 (compiled Thu Jun 26 11:08:16 2008)
# Network: main
# Input Vectors File: inputVector1.txt

```

```

.inputs f_X_Raw<0> f_X_Raw<1> f_X_Raw<2> f_X_Raw<3> f_X_Raw<4>
f_X_Raw<5> f_X_Raw<6> temp
.latches AA.Prev_th_Reset_Ini AA.state AA.timer BB.Prev_Pk_State
BB.f_X_Prev_PTSP<0> BB.f_X_Prev_PTSP<1> BB.f_X_Prev_PTSP<2>
BB.f_X_Prev_PTSP<3> BB.f_X_Prev_PTSP<4> BB.f_X_Prev_PTSP<5>
BB.f_X_Prev_PTSP<6> BB.f_X_t0<0> BB.f_X_t0<1> BB.f_X_t0<2>
BB.f_X_t0<3> BB.f_X_t0<4> BB.f_X_t0<5> BB.f_X_t0<6> CC.f_X_Prev<0>
CC.f_X_Prev<1> CC.f_X_Prev<2> CC.f_X_Prev<3> CC.f_X_Prev<4>
CC.f_X_Prev<5> CC.f_X_Prev<6> DD.state DD.th_Prev_X_Pretrip DD.timer
.outputs th_X_Pretrip
.initial 0 A0 T5 S1 0 0 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 S11 T0
.start_vectors

```

```

# f_X_Raw<0> f_X_Raw<1> f_X_Raw<2> f_X_Raw<3> f_X_Raw<4>
f_X_Raw<5> f_X_Raw<6> temp ; AA.Prev_th_Reset_Ini AA.state AA.timer
BB.Prev_Pk_State BB.f_X_Prev_PTSP<0> BB.f_X_Prev_PTSP<1>
BB.f_X_Prev_PTSP<2> BB.f_X_Prev_PTSP<3> BB.f_X_Prev_PTSP<4>
BB.f_X_Prev_PTSP<5> BB.f_X_Prev_PTSP<6> BB.f_X_t0<0> BB.f_X_t0<1>
BB.f_X_t0<2> BB.f_X_t0<3> BB.f_X_t0<4> BB.f_X_t0<5> BB.f_X_t0<6>
CC.f_X_Prev<0> CC.f_X_Prev<1> CC.f_X_Prev<2> CC.f_X_Prev<3>
CC.f_X_Prev<4> CC.f_X_Prev<5> CC.f_X_Prev<6> DD.state
DD.th_Prev_X_Pretrip DD.timer ; th_X_Pretrip

```

```

1 0 1 1 1 1 0 0 ; 0 A0 T5 S1 0 0 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 S11 T0 ; 1
1 0 1 1 1 1 0 1 ; 0 A0 T5 S1 0 0 1 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 1 1 S11 T1 ; 1
1 0 1 1 1 1 0 1 ; 1 A1 T0 S0 1 1 0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 S11 T2 ; 1
1 0 1 1 1 1 0 1 ; 0 A0 T1 S4 1 1 0 0 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1 1 1 S11 T3 ; 1
1 0 1 1 1 1 0 1 ; 0 A0 T2 S4 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 S11 T4 ; 1
1 0 1 1 1 1 0 1 ; 0 A0 T3 S4 1 1 0 0 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 1 S11 T5 ; 0
1 0 1 1 1 1 0 1 ; 0 A0 T4 S4 1 1 0 0 0 1 1 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 S00 T5 ; 0
1 0 1 1 1 1 0 0 ; 0 A0 T5 S4 1 1 0 0 0 1 1 0 0 0 1 0 1 1 0 0 0 1 0 1 1 1 S00 T5 ; 0
1 0 1 1 1 1 0 0 ; 0 A0 T5 S4 1 1 0 0 0 1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 S00 T5 ; 0
1 0 1 1 1 1 0 0 ; 0 A0 T5 S4 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 0 1 1 S00 T0 ; 0
1 0 1 1 1 1 0 0 ; 0 A0 T5 S4 1 1 0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 S00 T0 ; 0
1 0 1 1 1 1 0 0 ; 0 A0 T5 S4 1 1 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 1 0 1 S00 T0 ; 1
# Final State : 0 A0 T5 S4 1 1 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 1 0 1 S00 T0

```

Property & Result table

Property	Result
AG(((BB.BB.AAA.timer<0>=1 * BB.BB.AAA.timer<1>=0) * BB.BB.A...	Failed
AG(((BB.BB.AAA.timer<0>=1 * BB.BB.AAA.timer<1>=0) * BB.BB.A...	Failed
AG(((BB.BB.AAA.timer<0>=1 * BB.BB.AAA.timer<1>=0) * BB.BB.A...	Passed
AG(((BB.BB.AAA.timer<0>=1 * BB.BB.AAA.timer<1>=0) * BB.BB.A...	Passed

Flow chart

```

graph TD
    S0["BB.BB.AAA.timer:000  
BB.DD.th_Prev_X_Pretrip:1  
BB.EE.status:00  
BB.Prev.status:00"] -- f_X_Raw:1010000 --> S1["AA.f_X_Prev:1000111  
BB.BB.AAA.timer:001  
BB.DD.th_Prev_X_Pretrip:1  
BB.EE.status:01  
BB.Prev.status:01"]
    S1 -- f_X_Raw:1000111 --> S2["AA.f_X_Prev:1000111  
BB.BB.AAA.timer:010  
BB.DD.th_Prev_X_Pretrip:1  
BB.EE.status:01  
BB.Prev.status:01"]
    S2 -- f_X_Raw:1000111 --> S3["AA.f_X_Prev:1000111  
BB.BB.AAA.timer:011  
BB.DD.th_Prev_X_Pretrip:1  
BB.EE.status:01  
BB.Prev.status:01"]
    S3 -- f_X_Raw:1000111 --> S4["AA.f_X_Prev:1000111  
BB.BB.AAA.timer:100  
BB.DD.th_Prev_X_Pretrip:1  
BB.EE.status:01  
BB.Prev.status:01"]

```

Integer format Binary format

# state	input	File1Output	File2Output	File1State	File2State
0	Initial	Initial	Initial	S1 1 T0	S0 1 T0
1	f_X:61	1	1	S1 1 T1	S1 1 T1
2	f_X:61	1	1	S1 1 T2	S1 1 T2
3	f_X:61	1	1	S1 1 T3	S1 1 T3
4	f_X:61	1	1	S1 1 T4	S1 1 T4
5	f_X:61	1	1	S1 1 T5	S1 1 T5
6	f_X:61	0	0	S0 0 T5	S2 0 T5
7	f_X:52	0	0	S0 0 T0	S2 0 T0
8	f_X:52	1	0	Null	Null

VIS Analyzer 3.0

CBMC

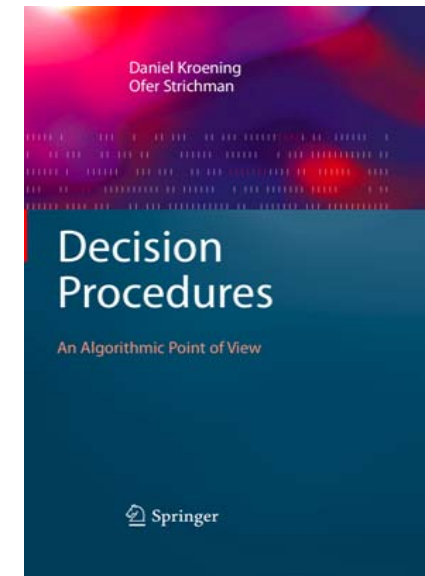
■ CBMC (C Bounded Model Checker)

¹⁾ <http://www.cprover.org/cbmc/>

A Bounded Model Checker for ANSI-C and C++ programs ¹⁾

- Verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions
- Checking **ANSI-C** and **C++** for consistency with other languages, such as Verilog
- Performing unwinding the loops in the program and passing the resulting equation to a decision procedure
- Supporting dynamic memory allocation using malloc and new

[SV Group Home](#)
[Software Verification](#)
[Hardware Verification](#)



CbmcSatabs - md2_bounds.c - Eclipse SDK

File Edit Refactor Navigate Search Project Run Window Help

Navigator

- demo
 - Results
 - .project
 - bounds.tsk
 - fptr4.tsk
 - int_overflow.tsk
 - md2_bounds.tsk
 - pointer_obj.tsk
 - pointer_to_local.tsk
 - small-c++.tsk
 - threads1.tsk
 - threads2.tsk

```

for (i = 0; i < 16; i++)
    x[i+32] = state[i] ^ block[i];

/* Encrypt block (18 rounds).
*/
t = 0;
for (i = 0; i < 18; i++) {
    for (j = 0; j < 48; j++)
        t = x[j] ^= PI_SUBST[t];
    t = (t + i) & 0xff;
}
  
```

Claims - SATABS - md2_bounds.tsk

File	Property	Description	Expression
R md2_bounds.c	bounds	array `x` upper bound	32 + i < 48
✓ md2_bounds.c	array bound	dereference failure: array `state` lower bound	!(i < 0) !(c::md2_bounds::MD2Tr
✓ md2_bounds.c	array bound	dereference failure: array `state` upper bound	!(c::md2_bounds::MD2Transform::
R md2_bounds.c	array bound	dereference failure: array `block` lower bound	!(i < 0) !(c::md2_bounds::MD2Tr
R md2_bounds.c	array bound	dereference failure: array `block` upper bound	!(c::md2_bounds::MD2Transform::
W md2_bounds.c	bounds	array `x` upper bound	TRUE
W md2_bounds.c	bounds	array `PI_SUBST` upper bound	t < 256
W md2_bounds.c	bounds	array `x` upper bound	TRUE
W md2_bounds.c	array bound	dereference failure: array `block` lower bound	!(i < 0) !(c::md2_bounds::MD2Tr
W md2_bounds.c	array bound	dereference failure: array `block` upper bound	!(c::md2_bounds::MD2Transform::
W md2_bounds.c	bounds	array `PI SUBST` upper bound	(t ^ (unsigned int)(*(i + block))) <

Trace Problems Log

```

Running Cadence SMV: smv -force -sift
Cadence SMV produced counterexample
Simulating abstract transitions of counterexample on concrete program
Spurious transition found
Trace is spurious
Refining transition
*** CEGAR Loop Iteration 6
Running Cadence SMV: smv -force -sift
  
```

CbmcSatabs - threads2.c - Eclipse SDK

File Edit Refactor Navigate Search Project Run Window Help

Navigator

- demo
 - Results
 - .project
 - bounds.tsk
 - fptr4.tsk
 - int_overflow.tsk
 - md2_bounds.tsk
 - pointer_obj.tsk
 - pointer_to_local.tsk
 - small-c++.tsk
 - threads1.tsk
 - threads2.tsk

```

md2_bounds.tsk
md2_bounds.c
threads2.tsk
threads2.c
g=1;
if (g==2)
{
    g=3;
    assert (g!=4);
}
}

int main()
{
    pthread_t id1, id2;

    pthread_create(&id1, NULL, t1, NULL);
    pthread_create(&id2, NULL, t2, NULL);
}
  
```

Claims - SATABS - threads2.tsk

Check Selection Check by File Check by Property Check All Stop Selection Stop All Stop Session Terminate Session Reset Session

File	Property	Description	Expression
✗ threads2.c	assertion	assertion	g != 4

Trace Problems Log

```

c::__CPROVER_alloc =
c::__CPROVER_alloc_size =
c::g = 0
c::t1::arg = NULL
c::g = 0
c::t2::arg = NULL
c::g = 1
c::g = 2
c::g = 3
c::g = 4
failure: assertion
  
```

Variable	Value	Type
c::t2::arg	NULL	void *
c::t1::arg	NULL	void *
c::__CPROVER_alloc		bool [INFINITY]
c::g	4	int
c::__CPROVER_alloc_size		unsigned int [INFINITY]

HyTech

■ HyTech

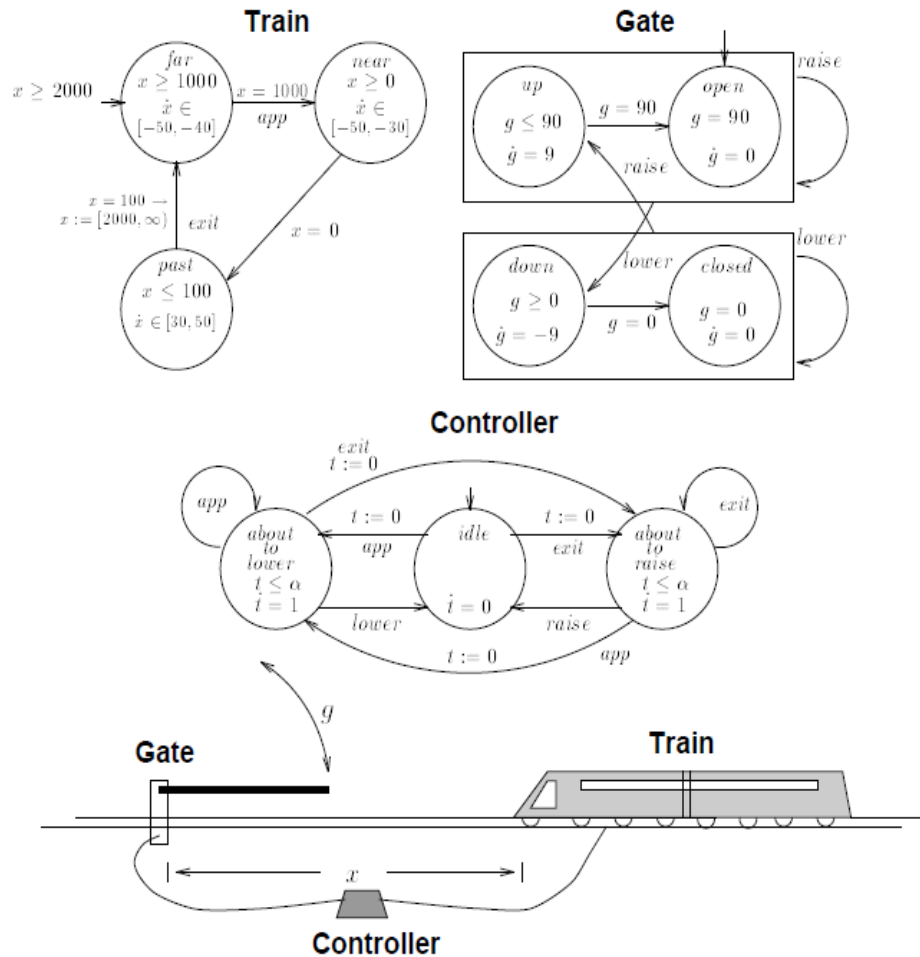
¹⁾ <http://embedded.eecs.berkeley.edu/research/hytech/>

An automatic tool for the analysis of a linear hybrid system with temporal requirements¹⁾

- Hybrid systems are specified as collections of **automata with discrete and continuous components**
- Temporal requirements are verified by **polyhedral model checking**
- Input: **Linear hybrid automata** (discrete + continuous variables)
+ closed system (no external input)
- No GUI



Railroad Crossing : Train, Controller, and Gate



$\alpha < 49/5$	Number of locations	Number of transitions	CPU time
When the train is within 10 meters to the gate, the gate is always fully closed.	36	90	0.2 sec.

```

Ubuntu [실행 중] - Oracle VM VirtualBox
jaeyeon@jaeyeon-VirtualBox: ~/HyTech_linux
HyTech: symbolic model checker for embedded systems
Version 1.04f (last modified 1/24/02) from v1.04a of 12/6/96
For more info:
  email: hytech@eecs.berkeley.edu
  http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a). Use -i for more info
=====
Number of iterations required for reachability: 9
===== Generating trace to specified target region =====
Time: 0.000
Location: closed.q.inflow_3
  on_off = 0 & contents = 0 & closing_time = 0 & 4inflow + 39 = 0 & t
= 0
-----
VIA:
-----
Time: 0.000
Location: closed.q1.inflow_3
  on_off = 1 & contents = 0 & closing_time = 0 & 4inflow + 39 = 0 & t
= 0
-----
VIA: on_off_e_1
-----
Time: 0.000
Location: open.q.inflow_3
  on_off = 1 & contents = 0 & closing_time = 0 & 4inflow + 39 = 0 & t
= 0
-----
VIA 9.750 time units
-----
Time: 9.750
Location: open.q.inflow_3
  on_off = 1 & 4contents = 39 & closing_time = 0 & inflow = 0 & 4t =
39
-----
VIA:
-----
  
```

SpaceEx

■ SpaceEx

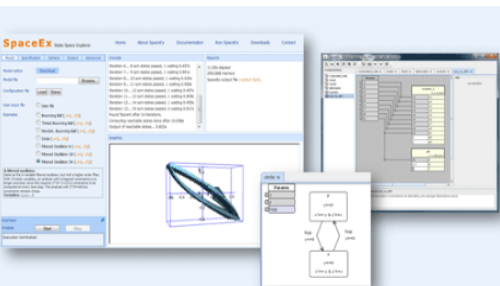
¹⁾ <http://spaceex.imag.fr/>

A tool platform is designed to facilitate the implementation of algorithms related to reachability and safety verification of hybrid systems ¹⁾

- Safety and reachability verification
- Input: **SX language** (Hybrid automata)
+ **Non-linear hybrid automata**
- Supporting External inputs
- But, hard to interpret counter-examples

SpaceEx State Space Explorer

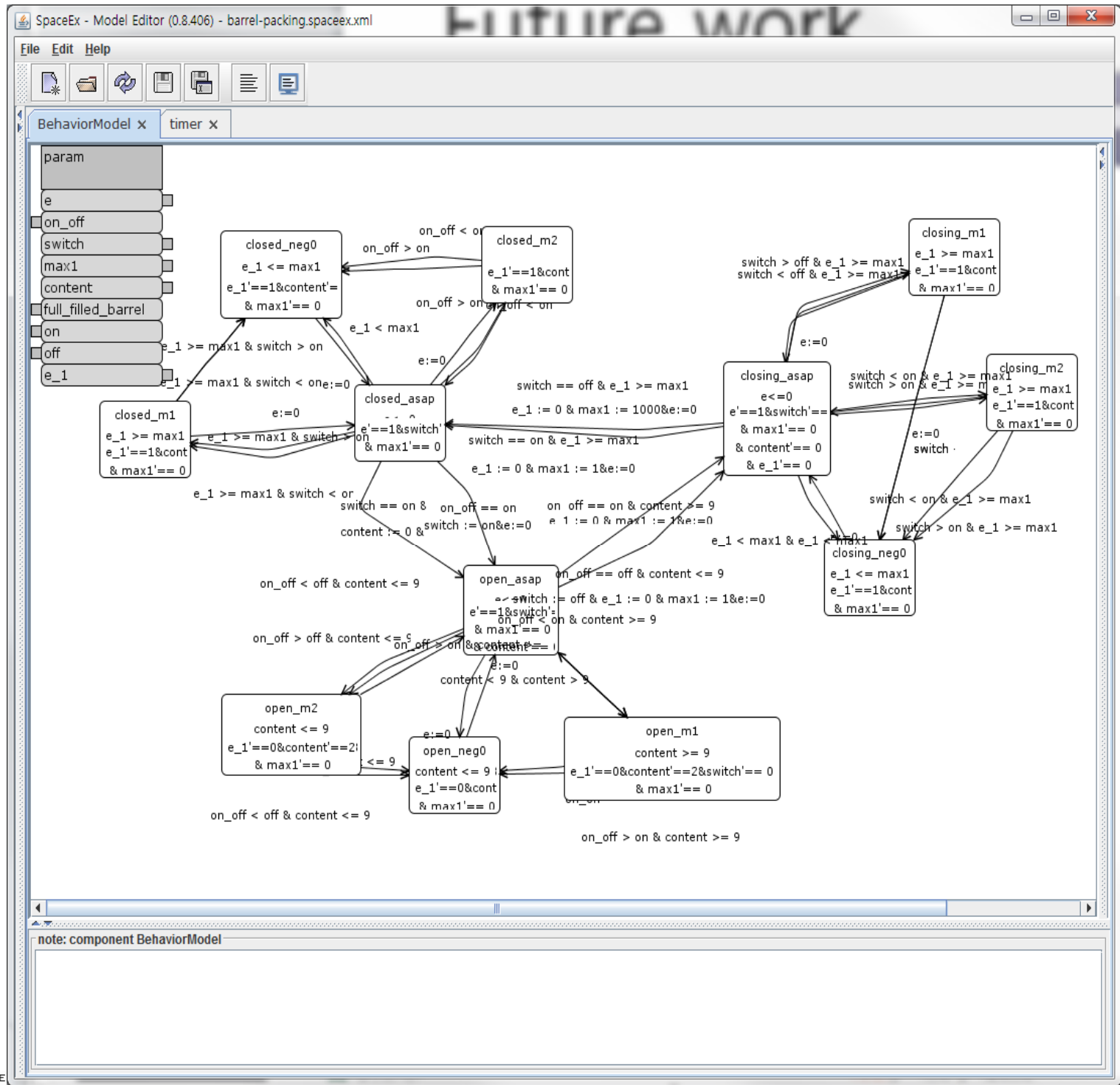
[Home](#)
[About SpaceEx](#)
[Documentation](#)
[Run SpaceEx](#)
[Downloads](#)



- Learn more about SpaceEx
- Download SpaceEx
- Subscribe to the newsletter

The verification of continuous and hybrid systems is a challenging problem, and various approaches are currently being investigated to overcome the complexities of representing and computing with continuous sets of states. Since verification problems are generally undecidable for such systems, experimental results are vital for evaluating and developing new ideas.

The SpaceEx tool platform is designed to facilitate the implementation of algorithms related to reachability and safety verification.



note: component BehaviorModel

SpaceEx [Virtual Machine Server]

[Home](#)
[About SpaceEx](#)
[Documentation](#)
[Run SpaceEx](#)
[Downloads](#)
[Contact](#)

Model Specification Options Output Advanced

System: Ctl_Fill_System

Ctl_Fill_System

- Controlled : Controller_1.e, valve, motor, position, level, Controller_1.e_sx, Filler_1.e, Filler_1.e_sx, Conveyer_1.e_sx, t
- Base-components : Controller_1, Filler_1, Conveyer_1, timer_1

Initial states

```
Controller_1.e==0 & valve==0 & motor==0 & position==0 & level==0 & Controller_1.e_sx==0 & Filler_1.e==0 & Filler_1.e_sx==0 & Conveyer_1.e_sx==0 & t==0 & loc(Controller_1)==init_asap & loc(Filler_1)==Stop_asap & loc(Conveyer_1)==Stop_asap
```

Forbidden states

```
level>10 || position >10
```

Analysis

Execution terminated

Console

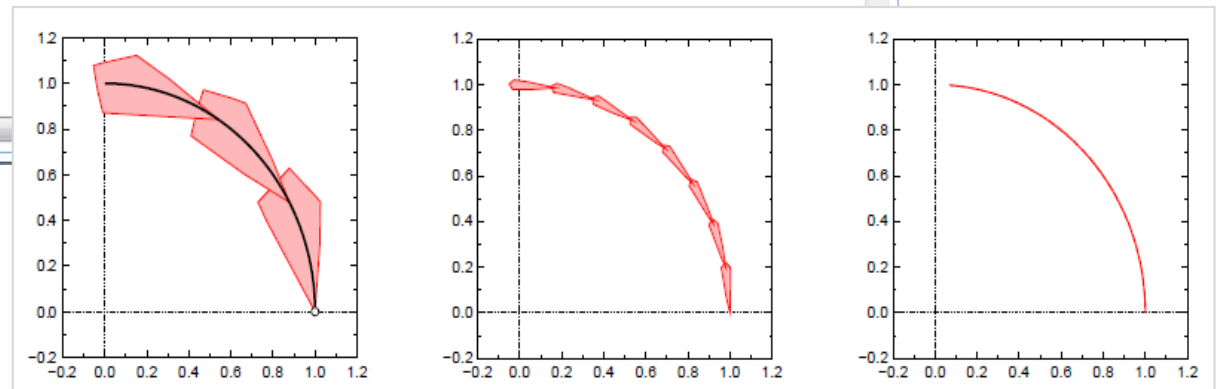
```
Iteration 555... 556 sym states passed, 3 waiting 0.004s
Iteration 556... 557 sym states passed, 2 waiting 0.003s
Iteration 557... 558 sym states passed, 1 waiting 0.003s
Iteration 558... 559 sym states passed, 2 waiting 0.002s
Iteration 559... 560 sym states passed, 2 waiting 0.005s
Iteration 560... 561 sym states passed, 1 waiting 0.004s
Iteration 561... 562 sym states passed, 0 waiting 0.004s
Found fixpoint after 562 iterations.
Computing reachable states done after 2.162s
Forbidden states are not reachable.
Output of reachable states... 0s
```

Reports

```
2.31s elapsed
2992KB memory
SpaceEx output file : output (txt).
```

Graphics

```
{ }
```



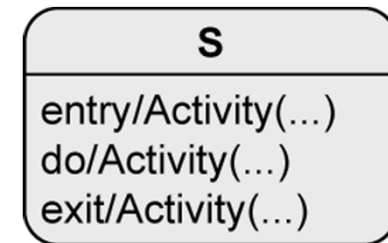
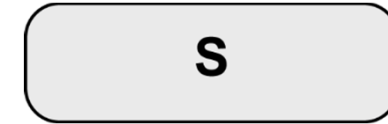
Statechart Diagram

Statechart Diagram

- Every object takes a finite number of different states during its life.
- **State machine (=Statechart) diagram** is used as follows:
 - to model the possible states of a system or object
 - to show how state transitions occur as a consequence of events
 - to show what behavior the system or object exhibits in each state

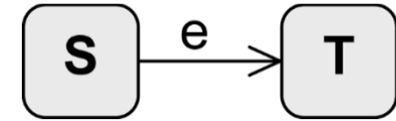
State

- **States** : nodes of the state machine
- When a state is active,
 - The object (or system) is in that state.
 - All internal activities specified in this state can be executed.
 - An activity can consist of multiple actions.
- State operations
 - entry / Activity(...)
 - Executed when the object enters the state
 - exit / Activity(...)
 - Executed when the object exits the state
 - do / Activity(...)
 - Executed while the object remains in this state

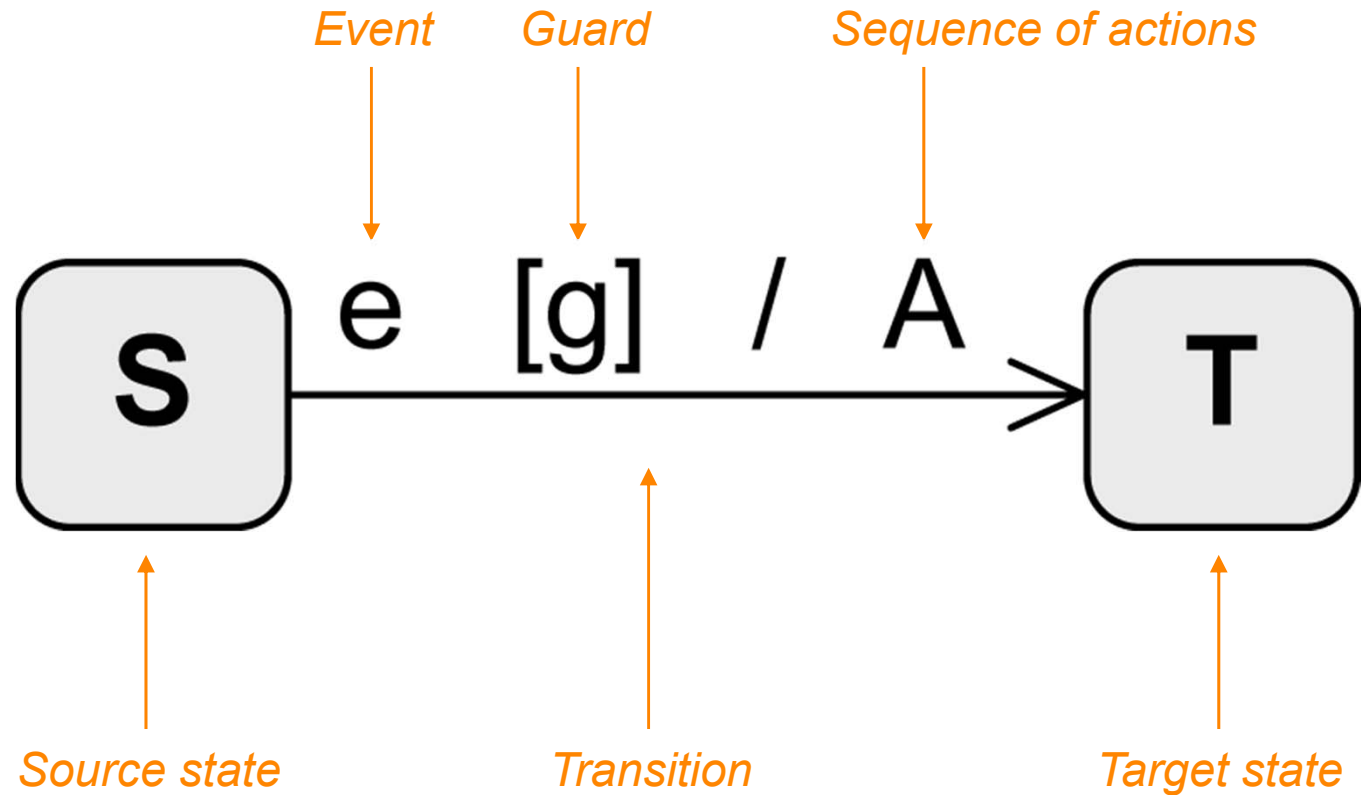


- : Initial state - Pseudostate
- : Final state - Real state
- ⊙ : Terminate node - Pseudostate
- ×

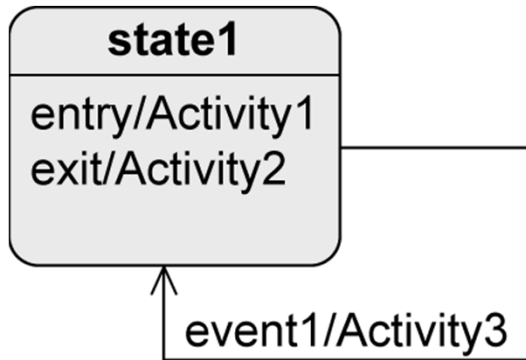
Transition



- Change from one state to another

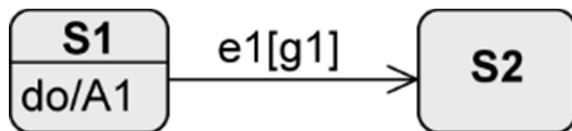


Transition : Examples

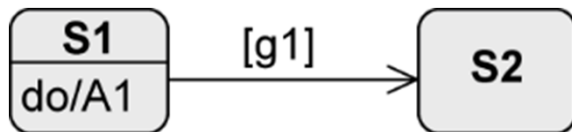


If **event1** occurs

- Object leaves `state1` and `Activity2` is executed
- `Activity3` is executed
- Object enters `state1` and `Activity1` is executed



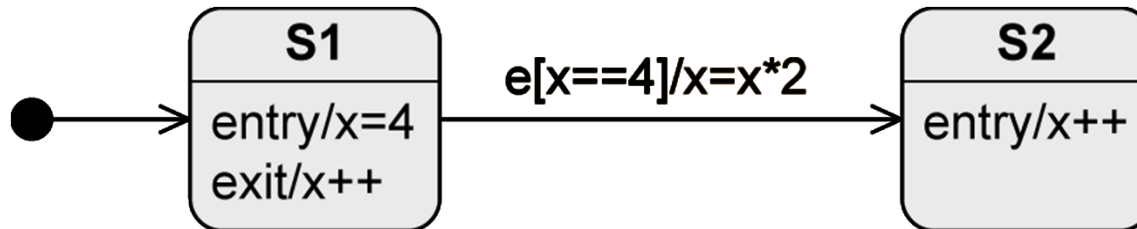
If **e1** occurs and **g1** evaluates to true, **A1** is aborted and the object changes to `s2`



As soon as the execution of **A1** is finished, a completion event is generated; if **g1** evaluates to true, the transition takes place; If not, this transition can never happen.

Transition - Sequence of Activity Executions

- Assume **s1** is active ... what is the value of **x** after **e** occurred?



s1 becomes active, **x** is set to the value **4**

e occurs, the guard is checked and evaluates to true

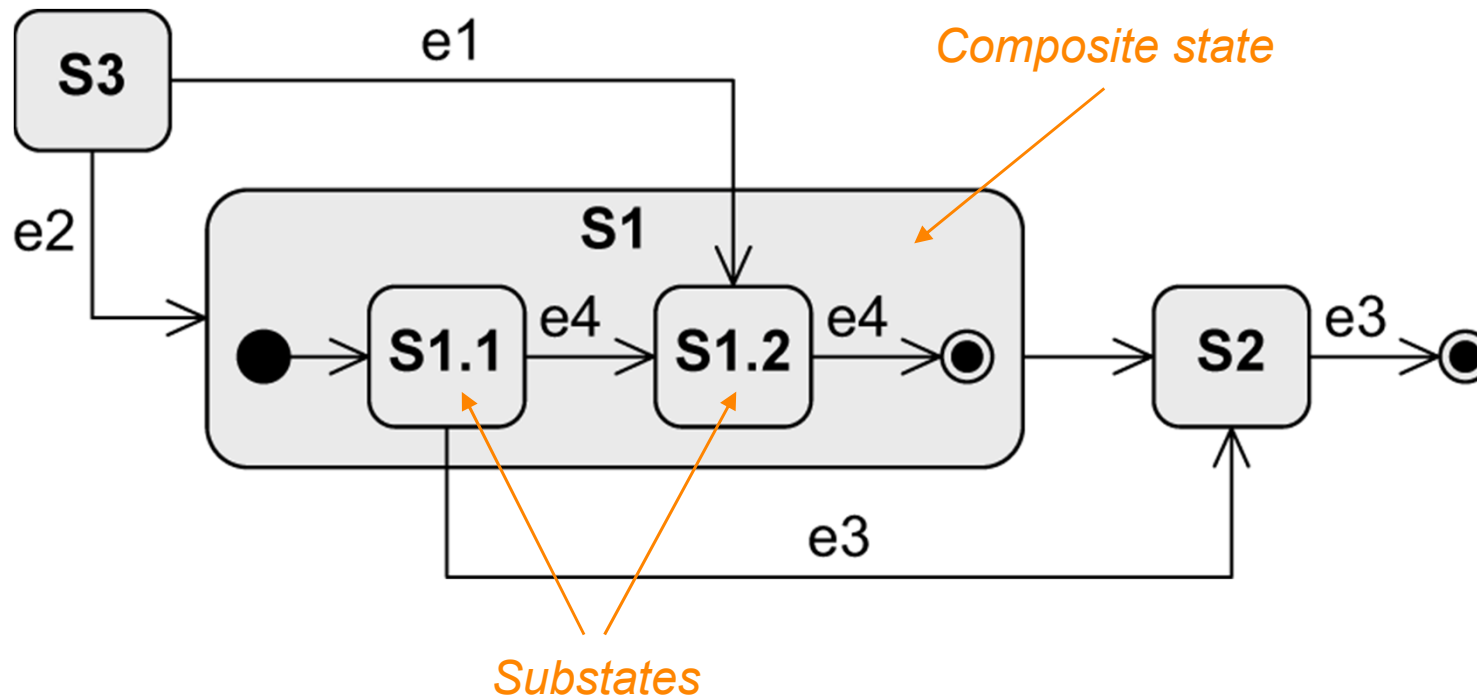
s1 is left, **x** is set to **5**

The transition takes place, **x** is set to **10**

s2 is entered, **x** is set to **11**

Composite State

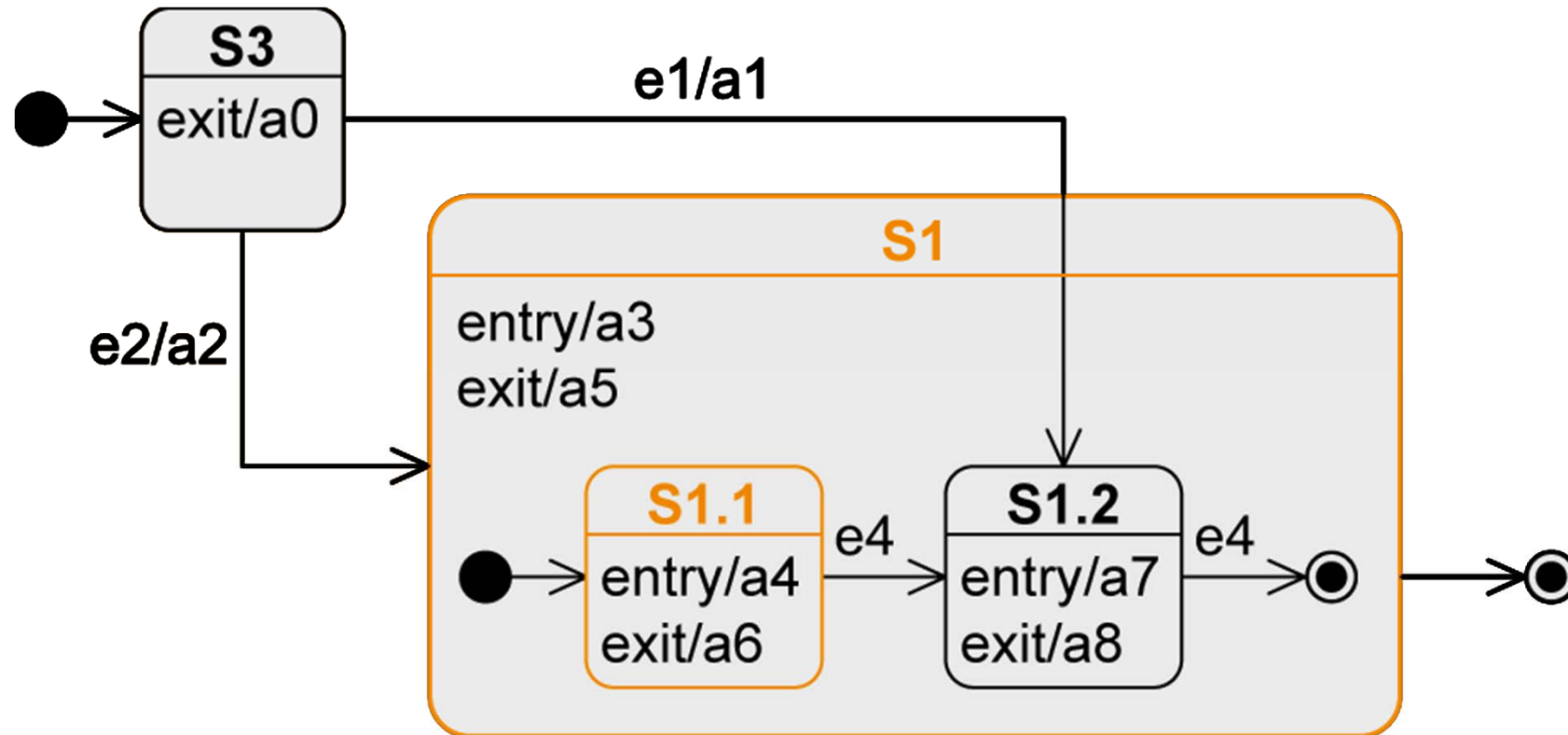
- Synonyms: complex state, nested state (→ **OR state**)
- Contains other states → “substates”
 - Only one of its substates is active at any point in time.
 - Arbitrary nesting depth of substates



Example : Entering a Composite State

- Transition to the boundary
 - Initial node of composite state is activated.

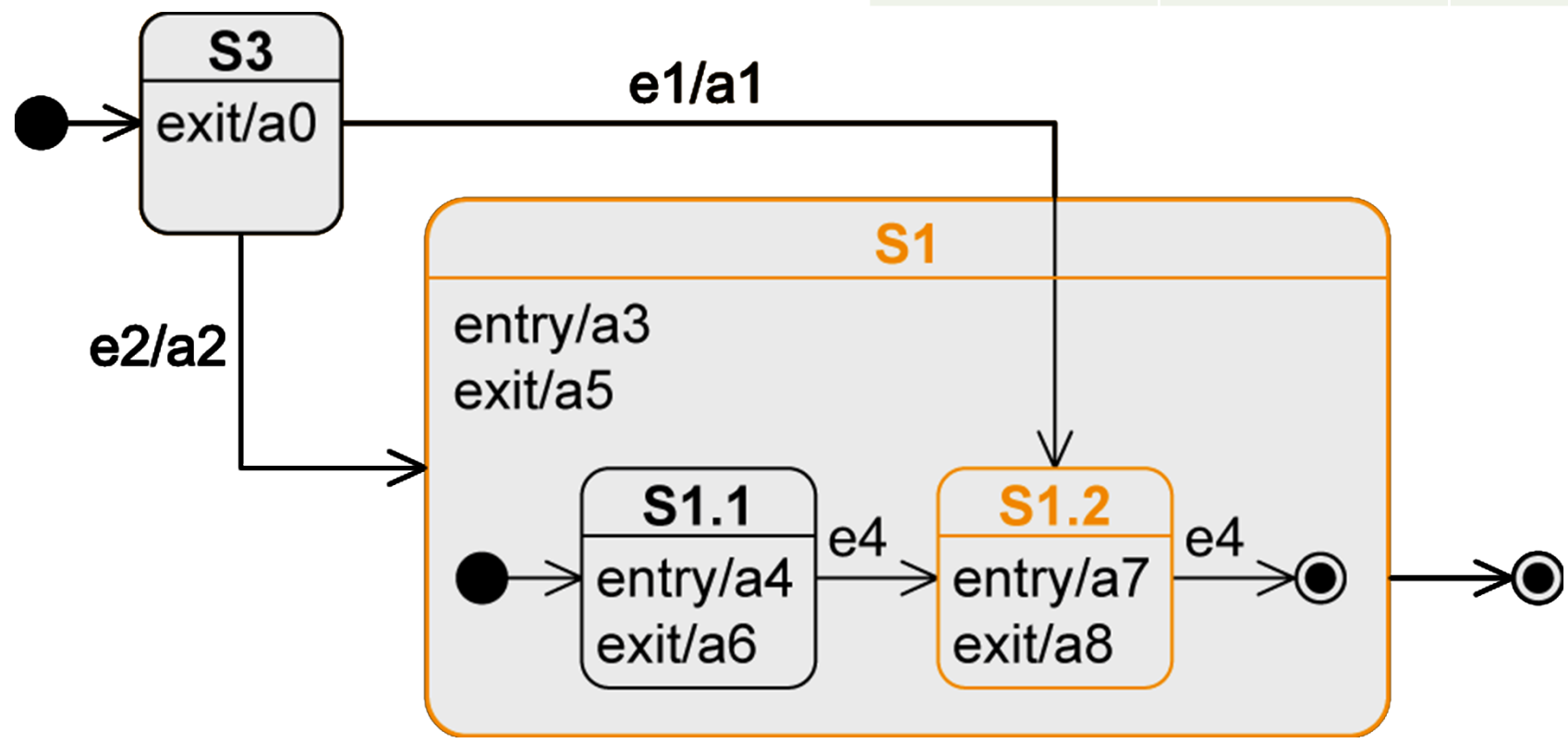
Event	State	Executed Activities
Beginning	S3	
e2	S1/S1.1	a0-a2-a3-a4



Example : Entering a Composite State

- Transition to a substate
 - Substate is activated.

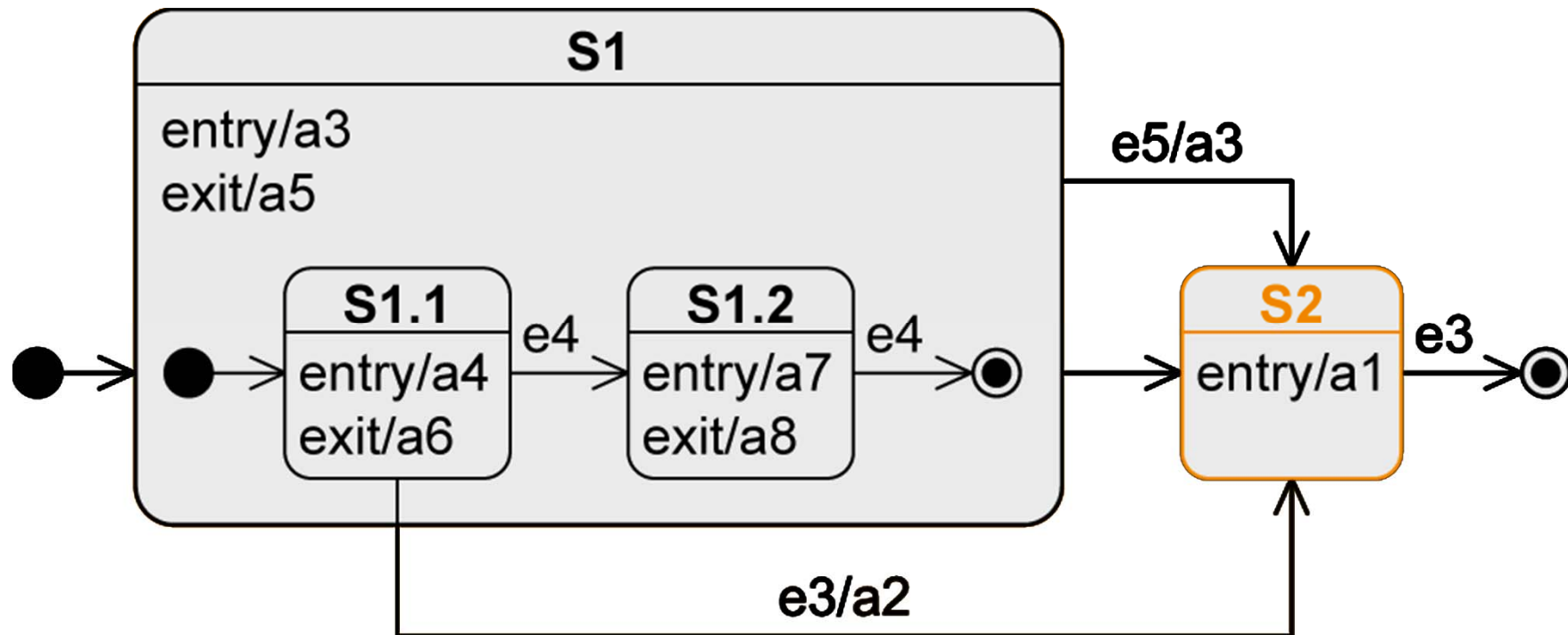
Event	State	Executed Activities
Beginning	S3	
e1	S1/S1.2	a0-a1-a3-a7



Example : Exiting from a Composite State

- Transition from a substate

Event	State	Executed Activities
Beginning	S1/S1.1	a3-a4
e3	S2	a6-a5-a2-a1

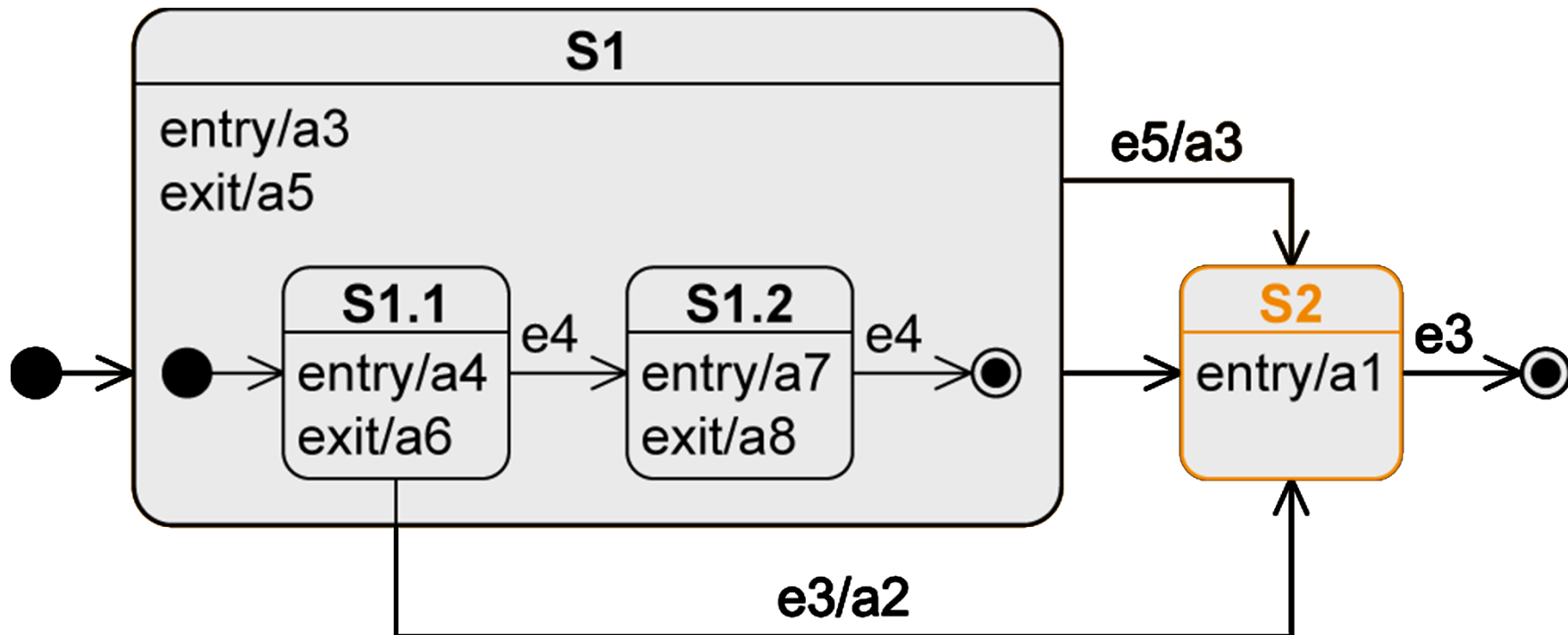


Example : Exiting from a Composite State

- Transition from the composite state

No matter which substate of S1 is active, as soon as e5 occurs, the system changes to S2

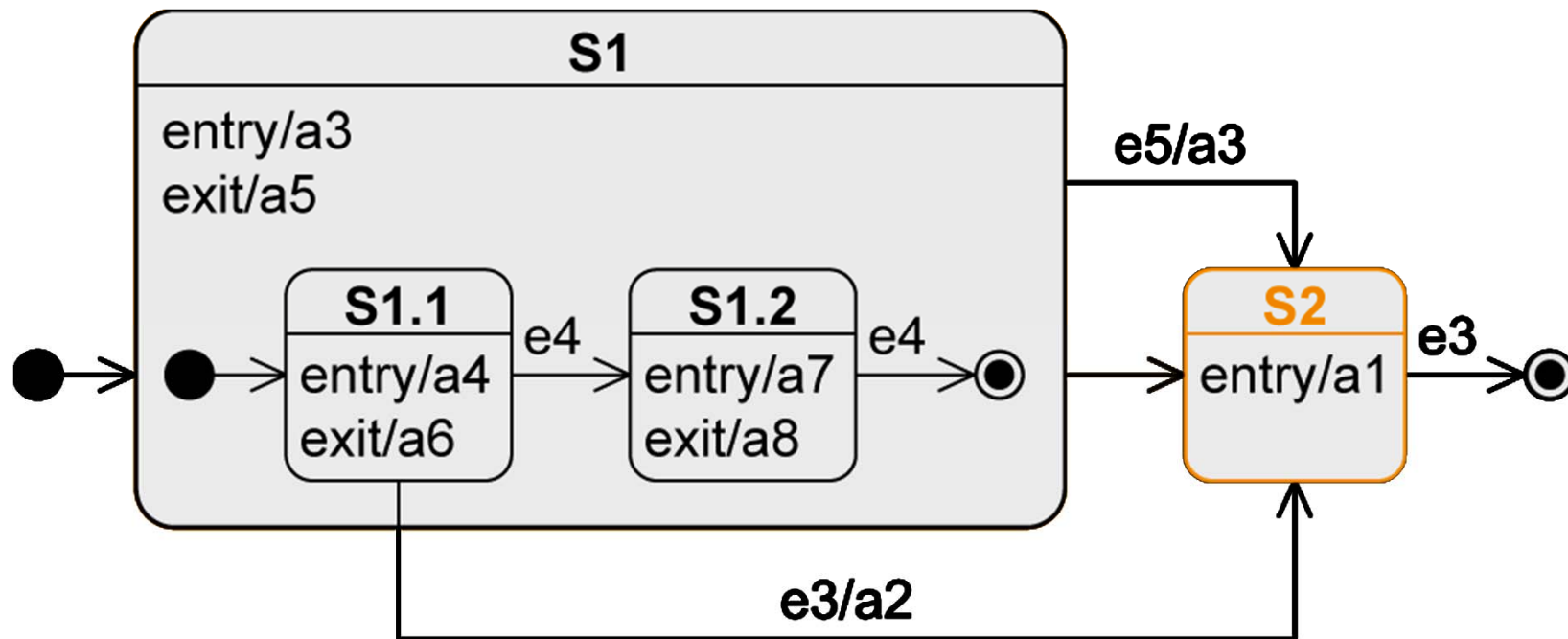
Event	State	Executed Activities
Beginning	S1/S1.1	a3-a4
e5	S2	a6-a5-a3-a1



Example : Exiting from a Composite State

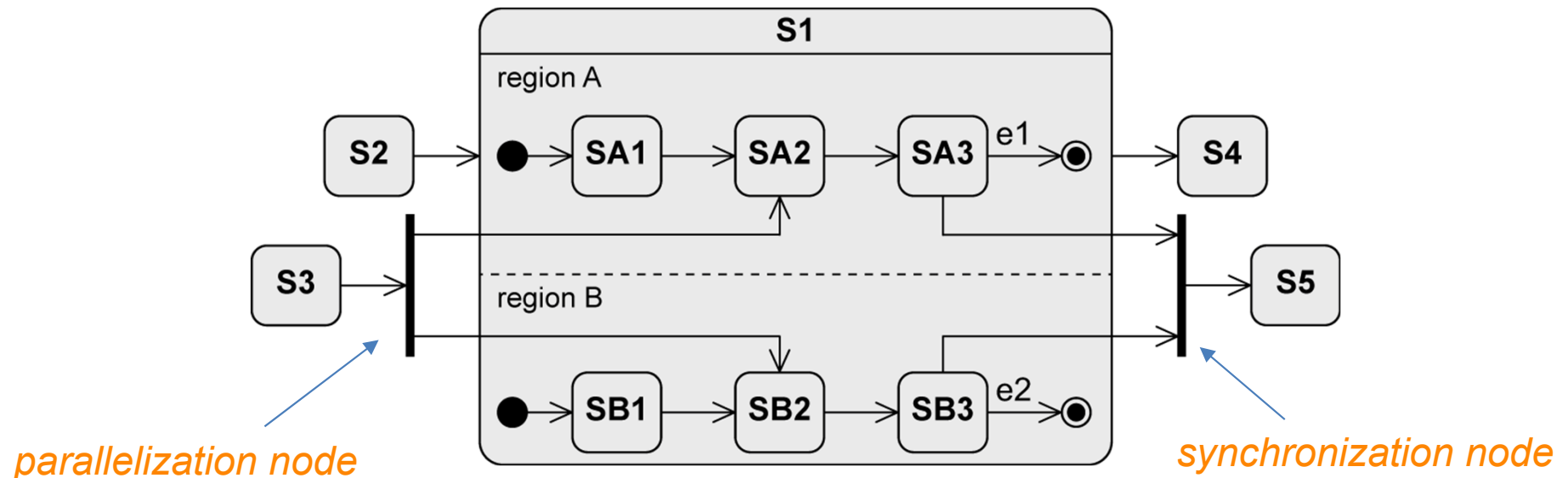
- Completion transition from the composite state

Event	State	Executed Activities
Beginning	S1/S1.1	a3-a4
e4	S1/S1.2	a6-a7
e4	S2	a8-a5-a1



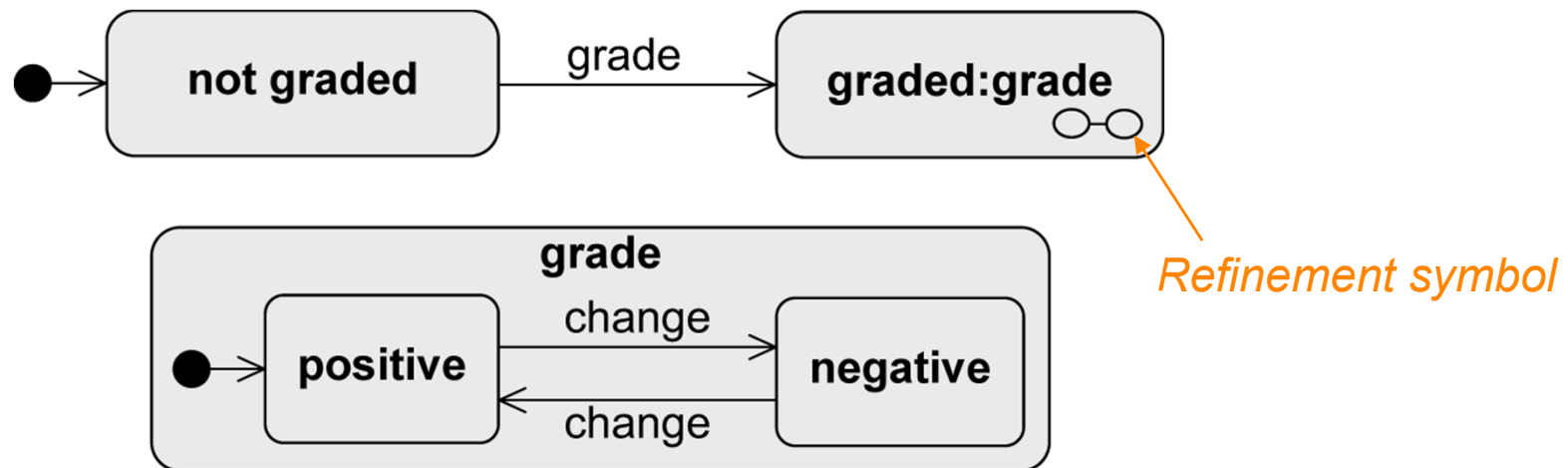
Orthogonal State

- Composite state is divided into two or more regions separated by a dashed line. (→ **AND State**)
 - One state of each region is always active at any point in time,
 - concurrent substates
- Entry: Transition to the boundary of the orthogonal state activates the initial states of all regions.
- Exit: Final state must be reached in all regions to trigger completion event.





Submachine State (SMS)

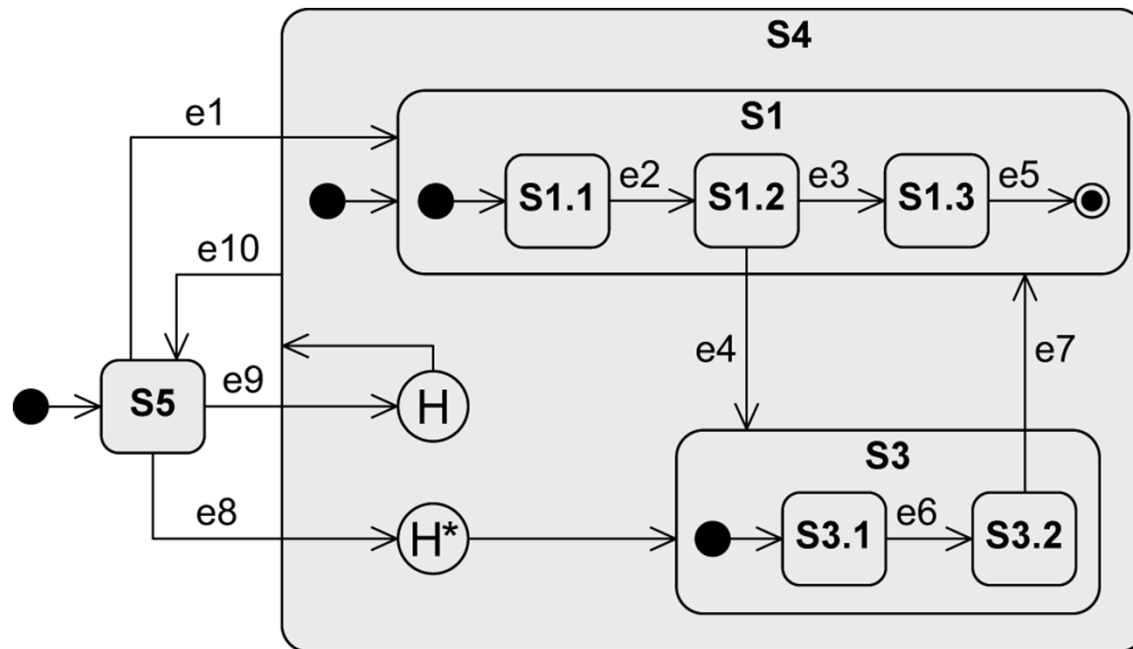
- To reuse parts of state machine diagrams in other state machine diagrams
 - Notation: `state:submachineState`
- As soon as the submachine state is activated, the behavior of the submachine is executed.
 - Corresponds to calling a subroutine in programming languages



History State

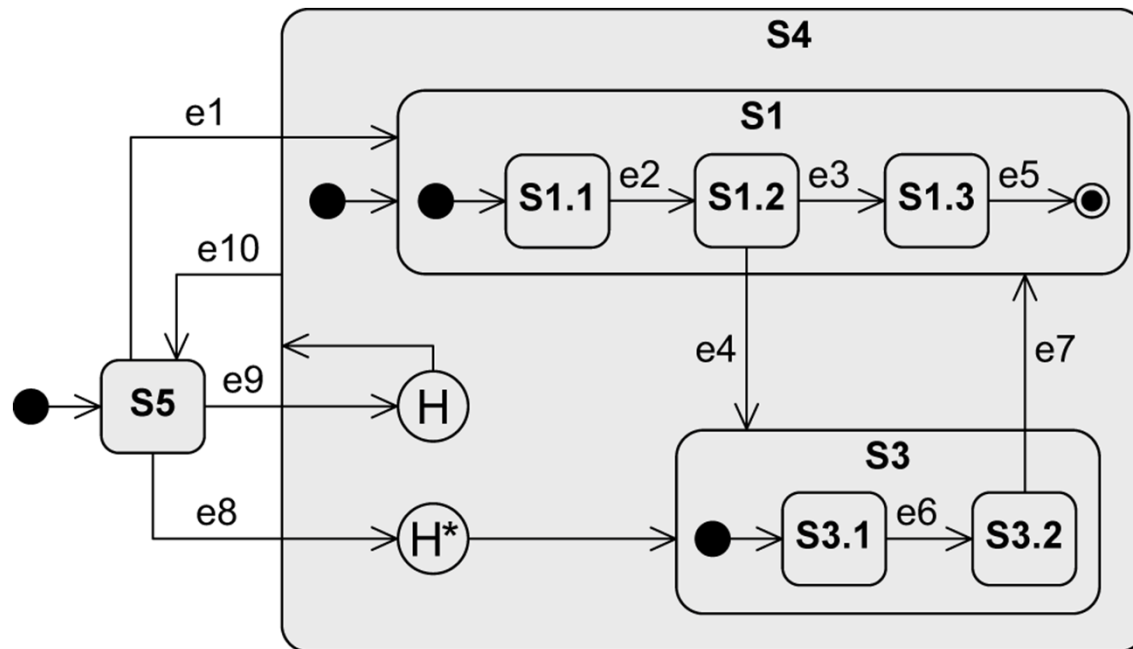
- To remembers which substate of a composite state was the last active one
 - Activates the “old” substate and all entry activities are conducted sequentially from the outside to the inside of the composite state
- **Shallow history** state restores the state that is on the same level of the composite state. 
- **Deep history state** restores the last active substate over the entire nesting depth. 

Example: History State



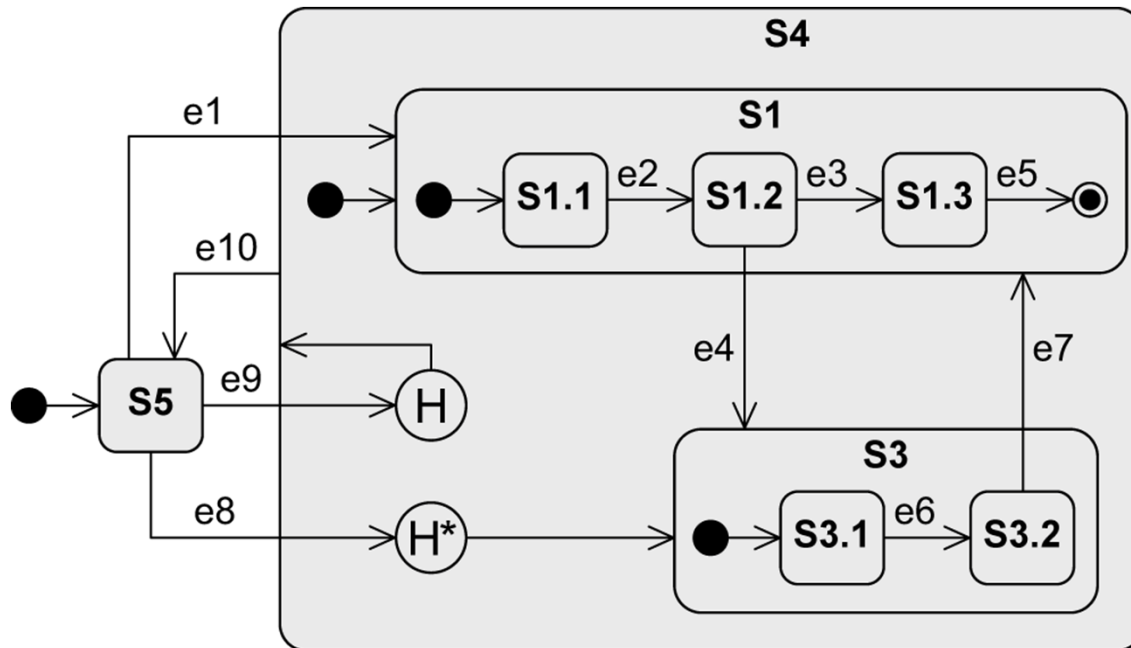
Event	State
Beginning	S5
e1	S4/S1/S1.1
e2	S1.2
e10	S5
e9	(H→) S1/S1.1

Example: History State



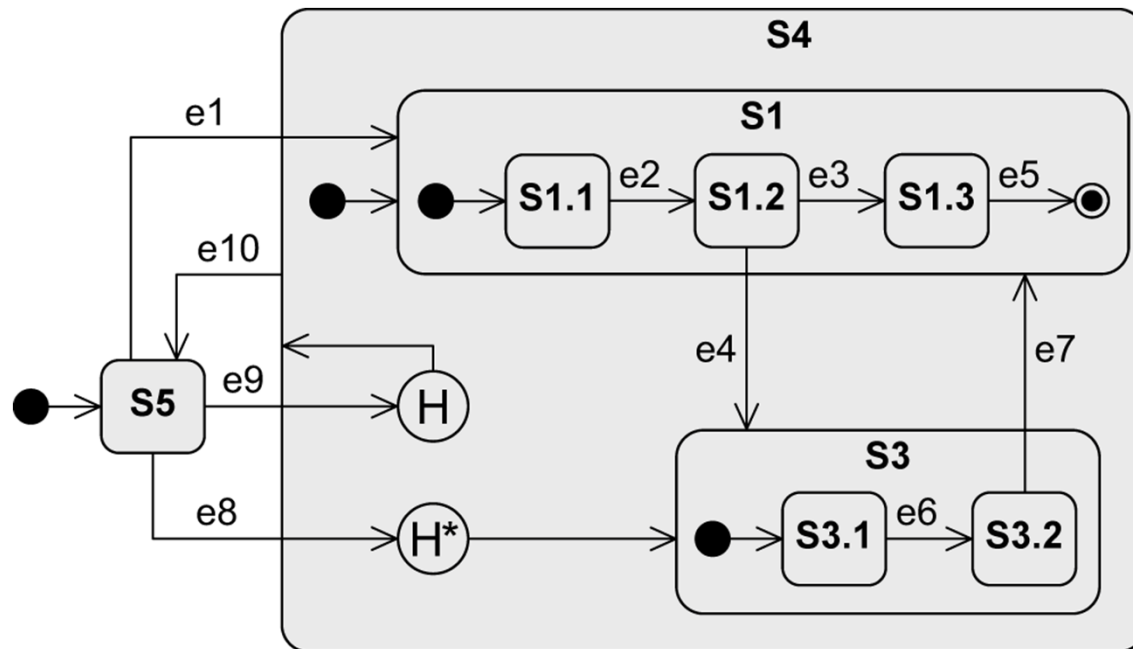
Event	State
Beginning	S5
e1	S4/S1/S1.1
e2	S1.2
e10	S5
e8	(H* →) S1.2

Example: History State



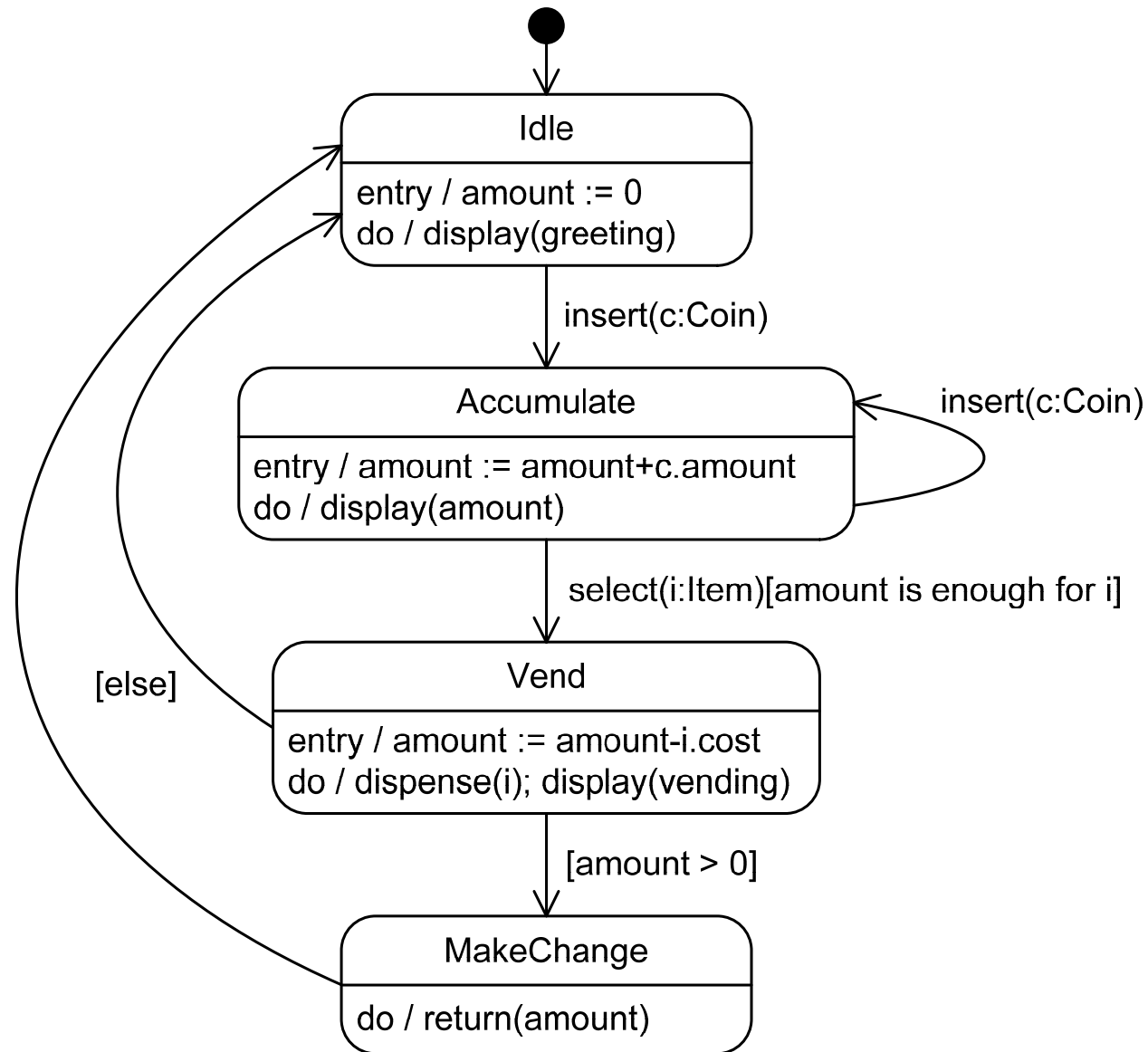
Event	State
Beginning	S5
e9	(H→) S1/S1.1

Example: History State

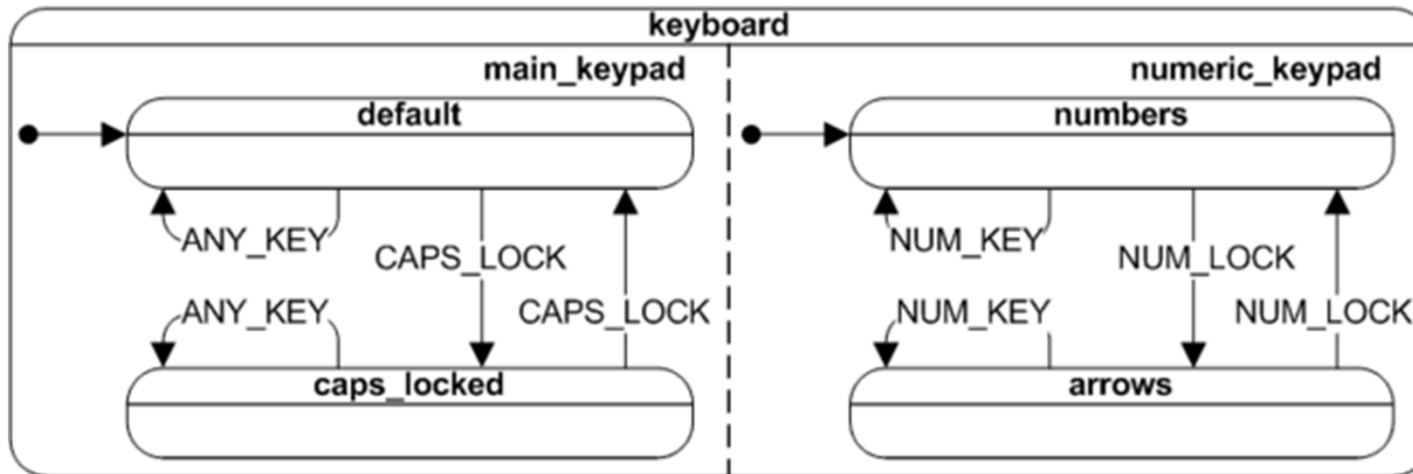


Event	State
Beginning	S5
e8	(H* →) S3/S3.1

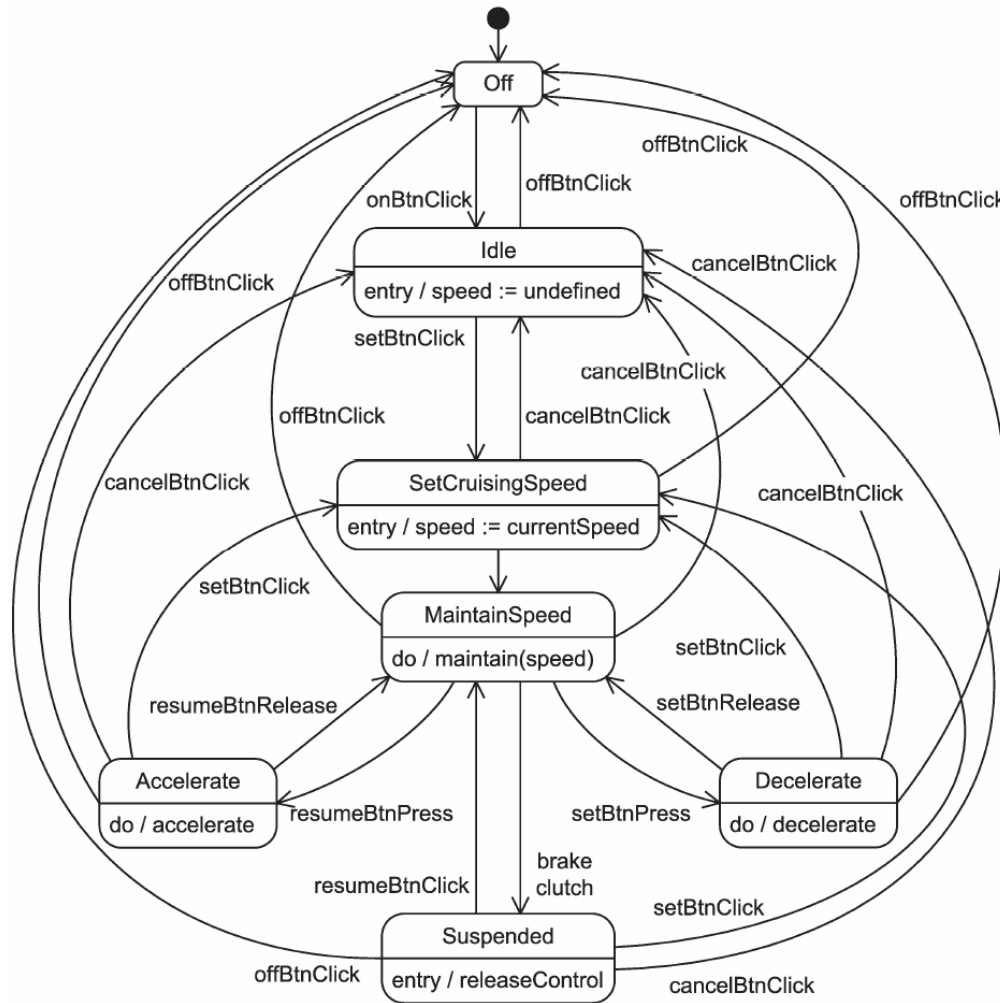
More Examples : Vending Machine



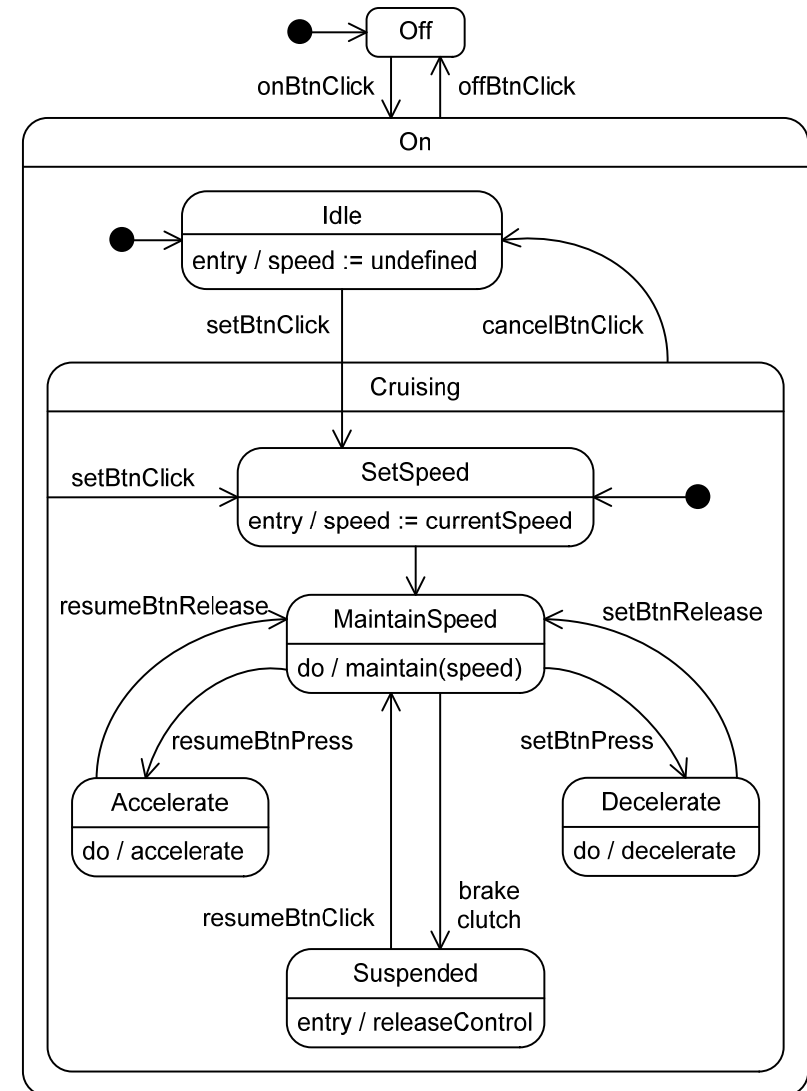
More Examples : Keyboard



More Examples : Cruise Control System



=



Statecharts Modeling

Statechart Modeling - CVM

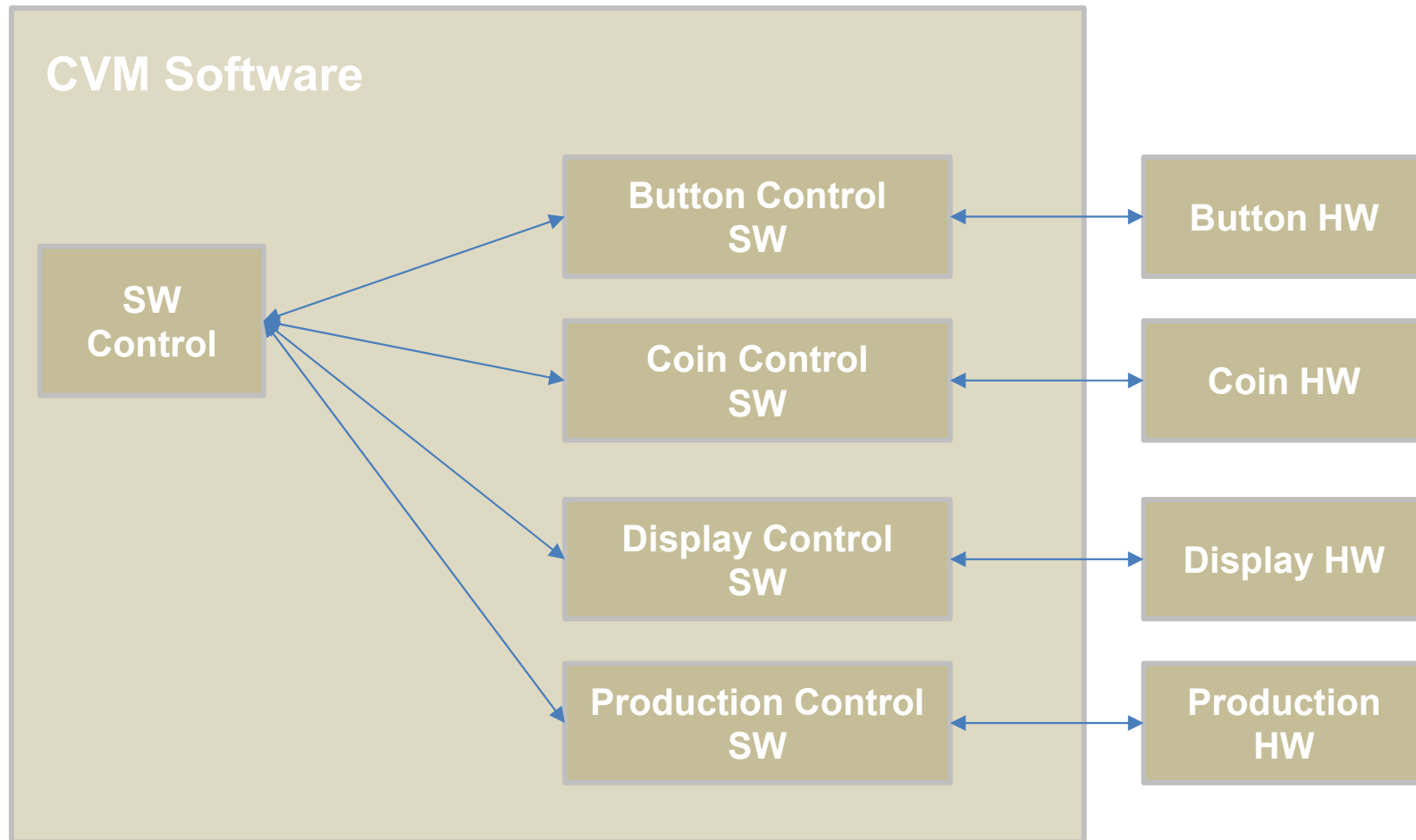
- Let's perform the Statechart modeling for CVM (Coffee Vending Machine).
 - Consisting of one or several control SW(s) and various HWs

- Modeling tool:
 - StarUML
 - YAKINDU Statecharts
 - 종이와 연필

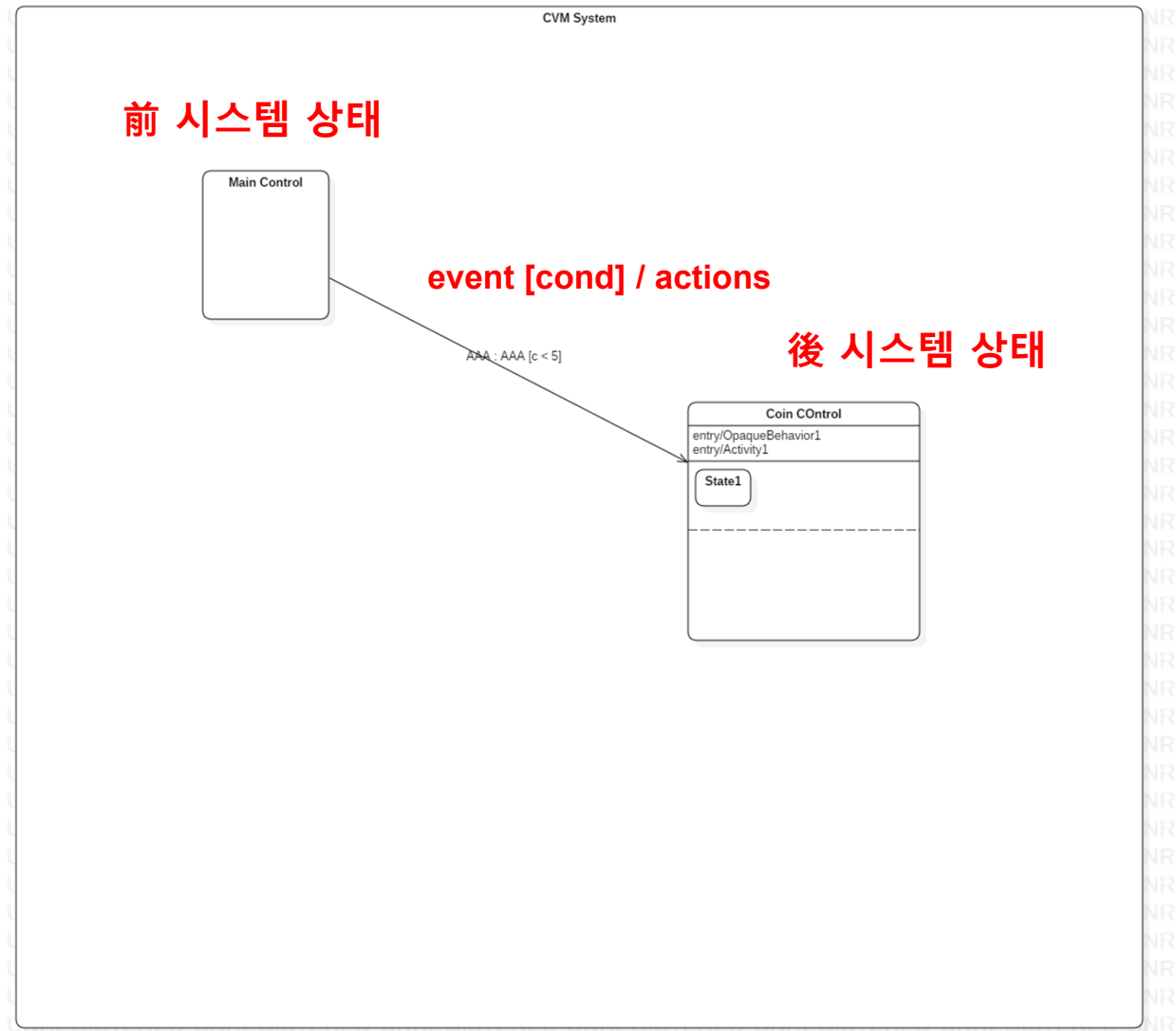
- Modeling features
 - Common CVMs
 - 5 different coffees with different value : input
 - Coins : 50, 100, 500, 1000 : input
 - Refund : output
 - Out-of-services : output
 - Vending : output



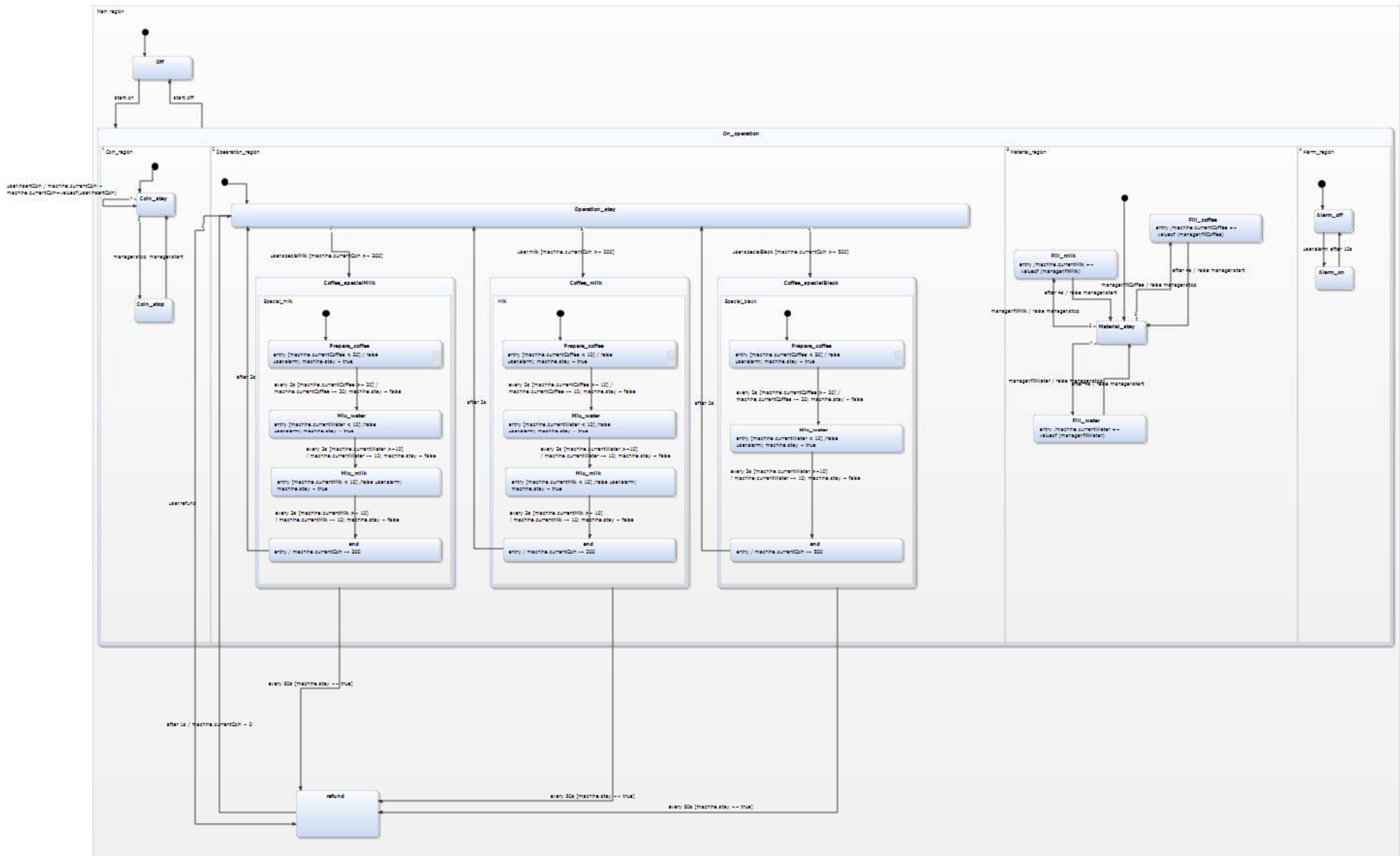
CVM - A Draft Architecture



CVM – A StarUML Example



Example : 커피 자판기 모델 (YAKINDU Statecharts)



YAKINDU Statechart Tool

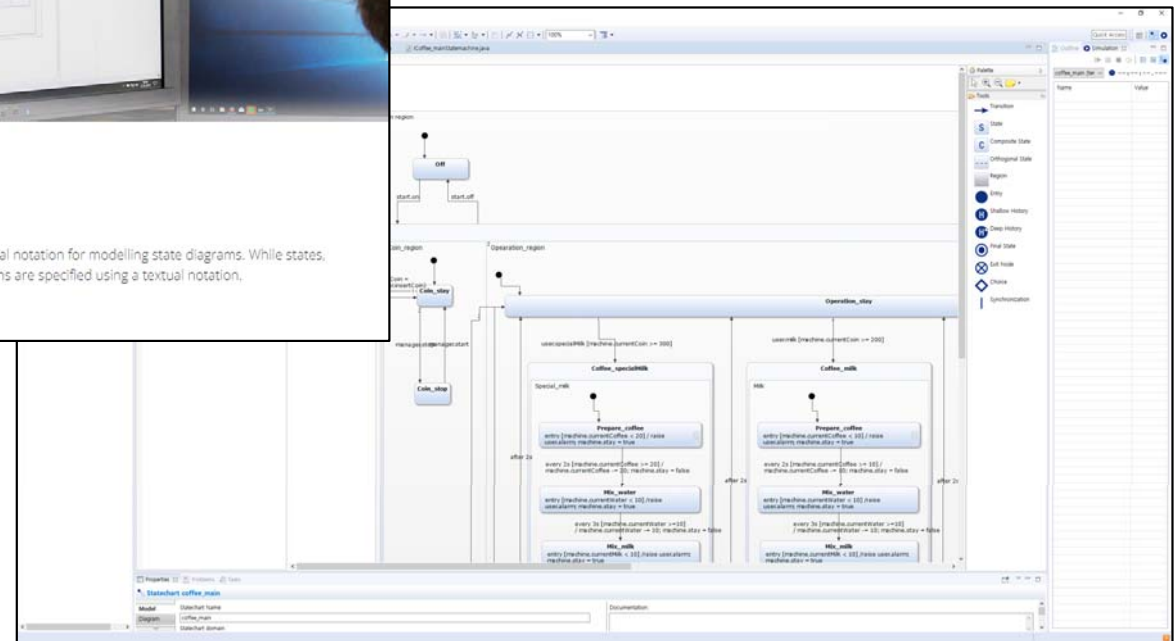
- YAKINDU statechart tool
 - <https://www.itemis.com/en/yakindu/state-machine/>

The screenshot shows the website for YAKINDU Statechart Tools. At the top, there is a navigation menu with links for Services, Products, Company, Blog, Career, and Contact. Below this is a secondary menu with Home, Documentation, Licenses, Training, Community, and Download. A prominent banner features the text "Use the power of state machines" and "YAKINDU Statechart Tools provides an integrated modelling environment for the specification and development of reactive, event-driven systems based on the concept of state machines." Two buttons are visible: "DOWNLOAD NOW" and "BUY A LICENSE". A "Free live webinar" section with a "REQUEST" button is also present.

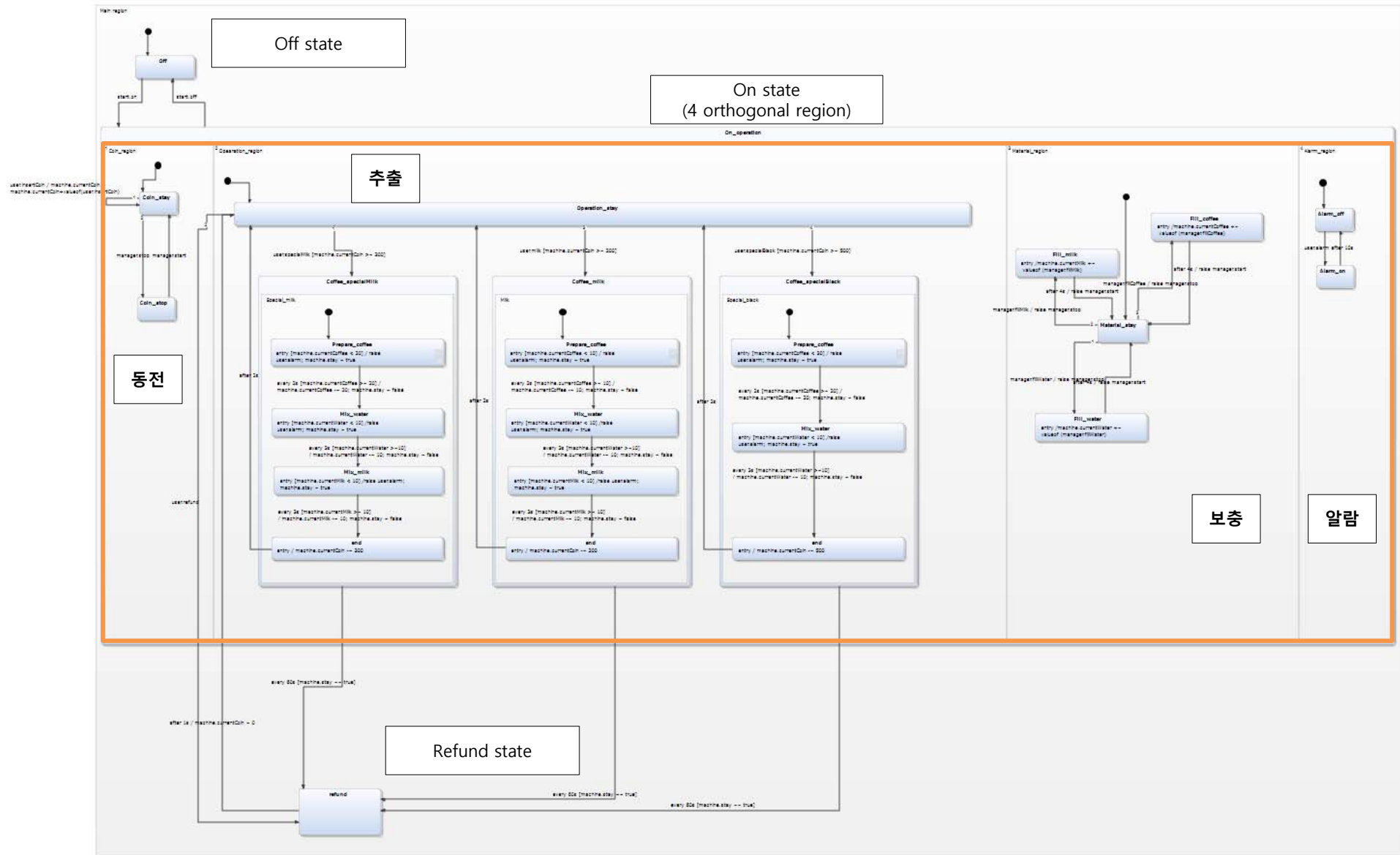


Editing

YAKINDU Statechart Tools features an intuitive combination of graphical and textual notation for modelling state diagrams. While states, transitions and state hierarchies are graphical elements, all declarations and actions are specified using a textual notation. The usability of the statechart editor is simply fascinating.



Example : 커피 자판기 모델 (YAKINDU Statecharts)



커피 자판기 모델

- It consists of 2 state and 1 orthogonal state (with 4 region)
 - 1. Off state
 - 2. Refund state
 - 3. Operation state (orthogonal state)
 - 동전 (Coin_region)
 - 추출 (Operation_region)
 - 보충 (Material_region)
 - 알람 (Alarm_region)

커피 자판기 모델

- Definition of the event and variable
 - Each event has a valuable data type

```
@CycleBased(200)
```

```
interface start:
  in event on
  in event off
```

```
interface user:
  in event insertCoin : integer
  in event refund
  in event specialMilk
  in event milk
  in event specialBlack
  in event alarm
```

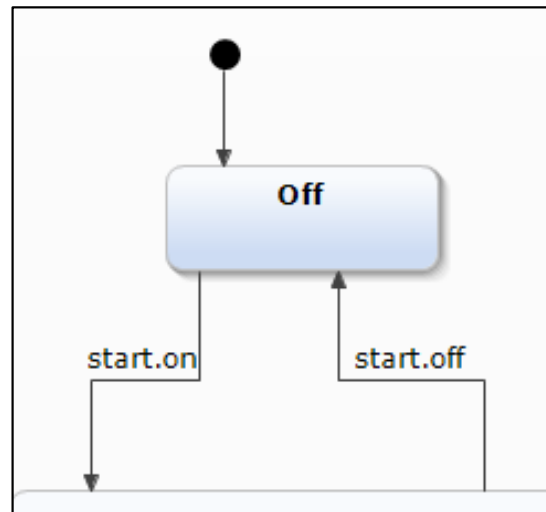
```
interface machine:
  var currentCoin : integer
  var currentMilk : integer = 1000
  var currentWater : integer = 1000
  var currentCoffee : integer = 500
  var stay : boolean = false
```

```
interface manager:
  in event fillWater : integer
  in event fillCoffee : integer
  in event fillMilk : integer
  in event stop
  in event start
```

```
// Define events and
// and variables here.
```

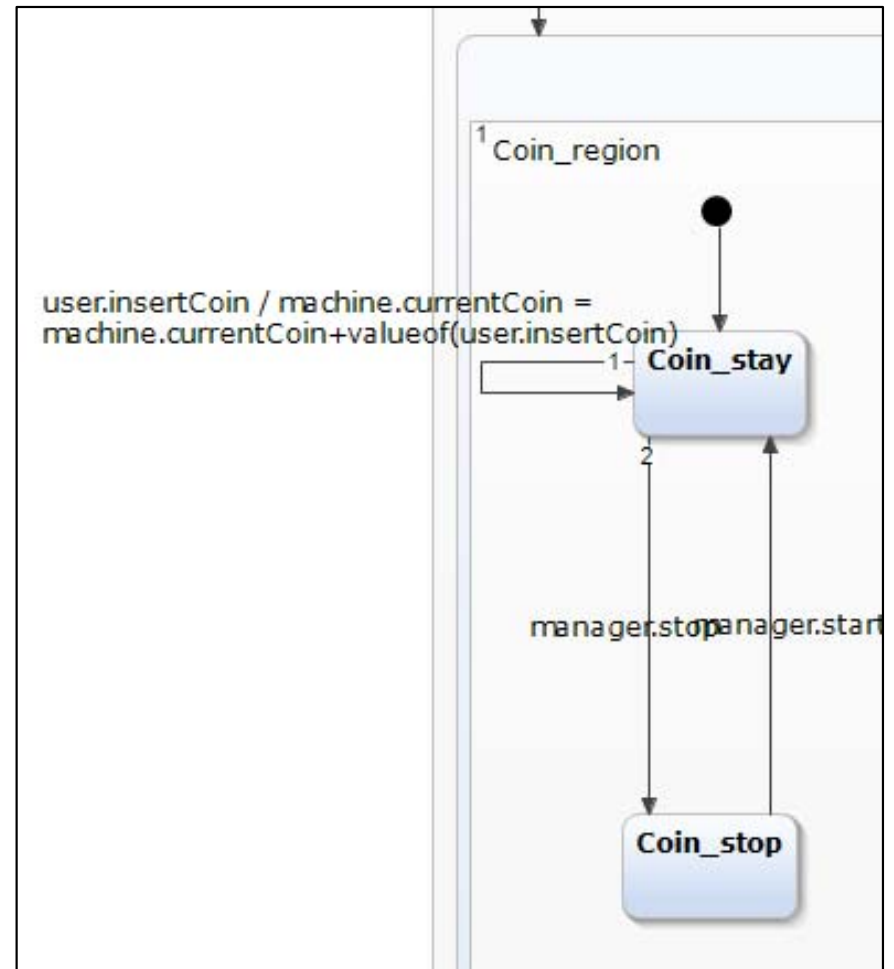
Off state

- 자판기의 off 상태 표시
 - Start event를 통해서 off/operation state로의 전환 담당



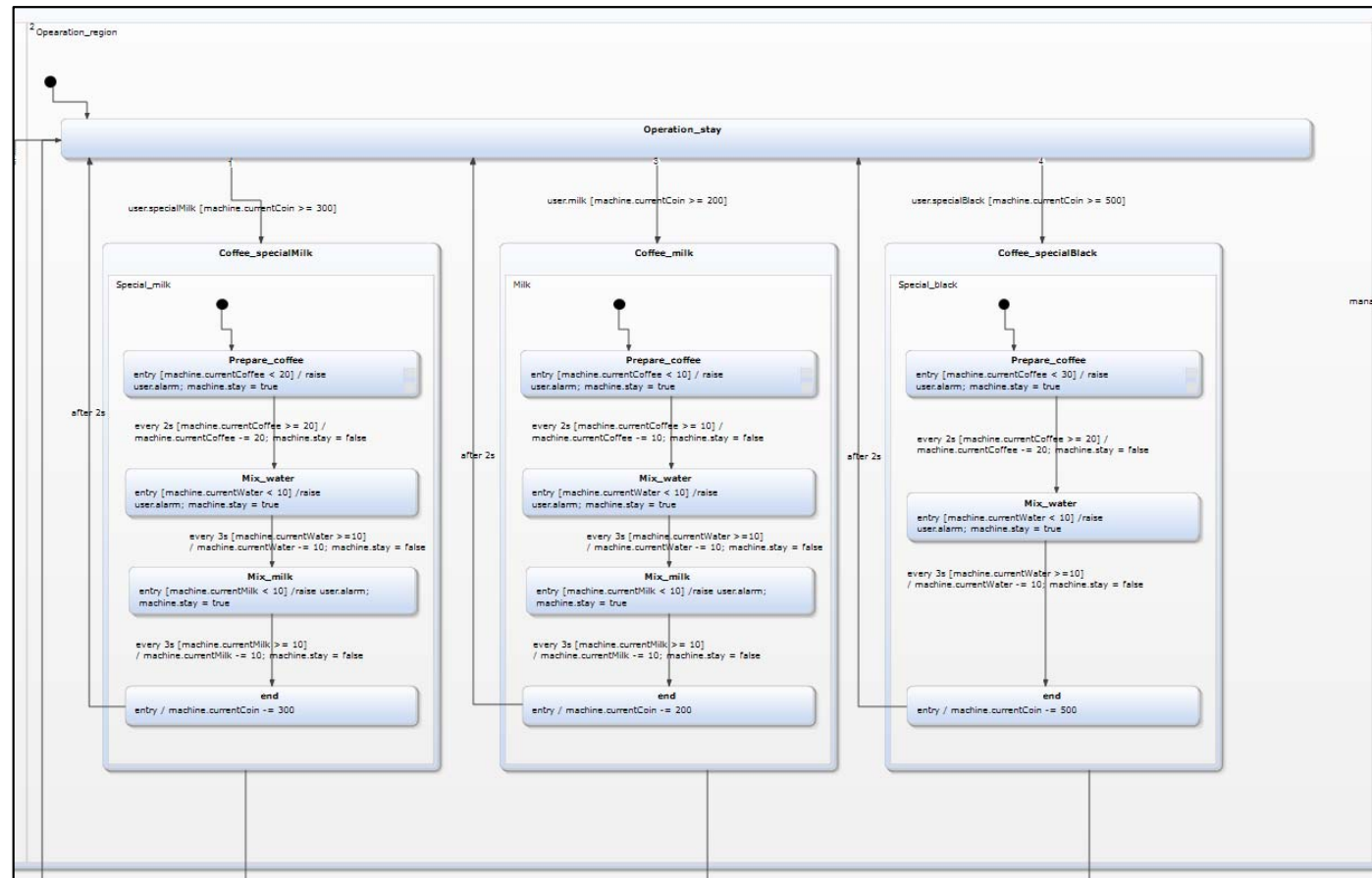
On_operation state - Coin_region

- 자판기 사용을 위한 동전 삽입 region
 - Stay 상태에서 언제든지 삽입 가능



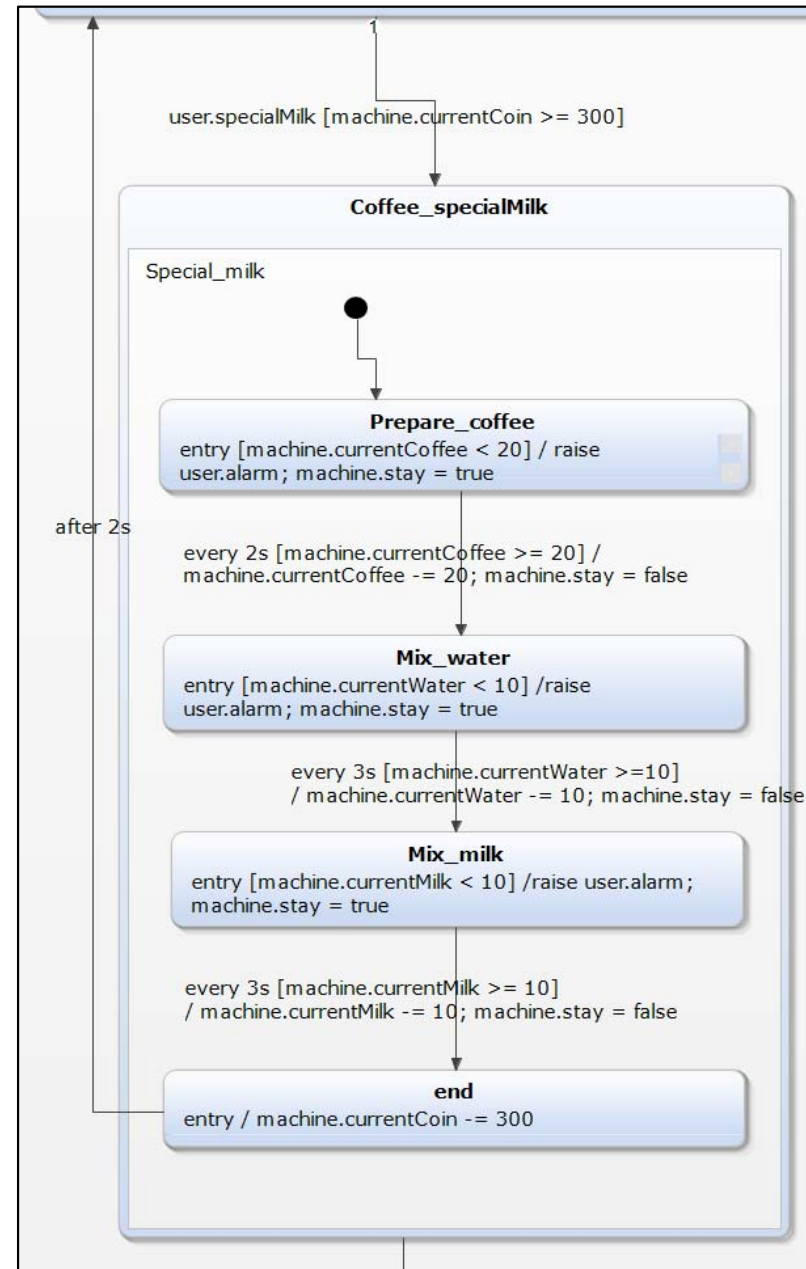
On_operation state - Operation_region

- 자판기 에서 추출을 위한 부분
 - 3 종류의 커피 추출 (specialMilk, milk, specialBlack)
 - 추출 이벤트에 따라 재료 확인 후 일정 시간 후 추출
 - 재료 부족 시 알람 이벤트 및 일정 시간 이후에 환불 진행



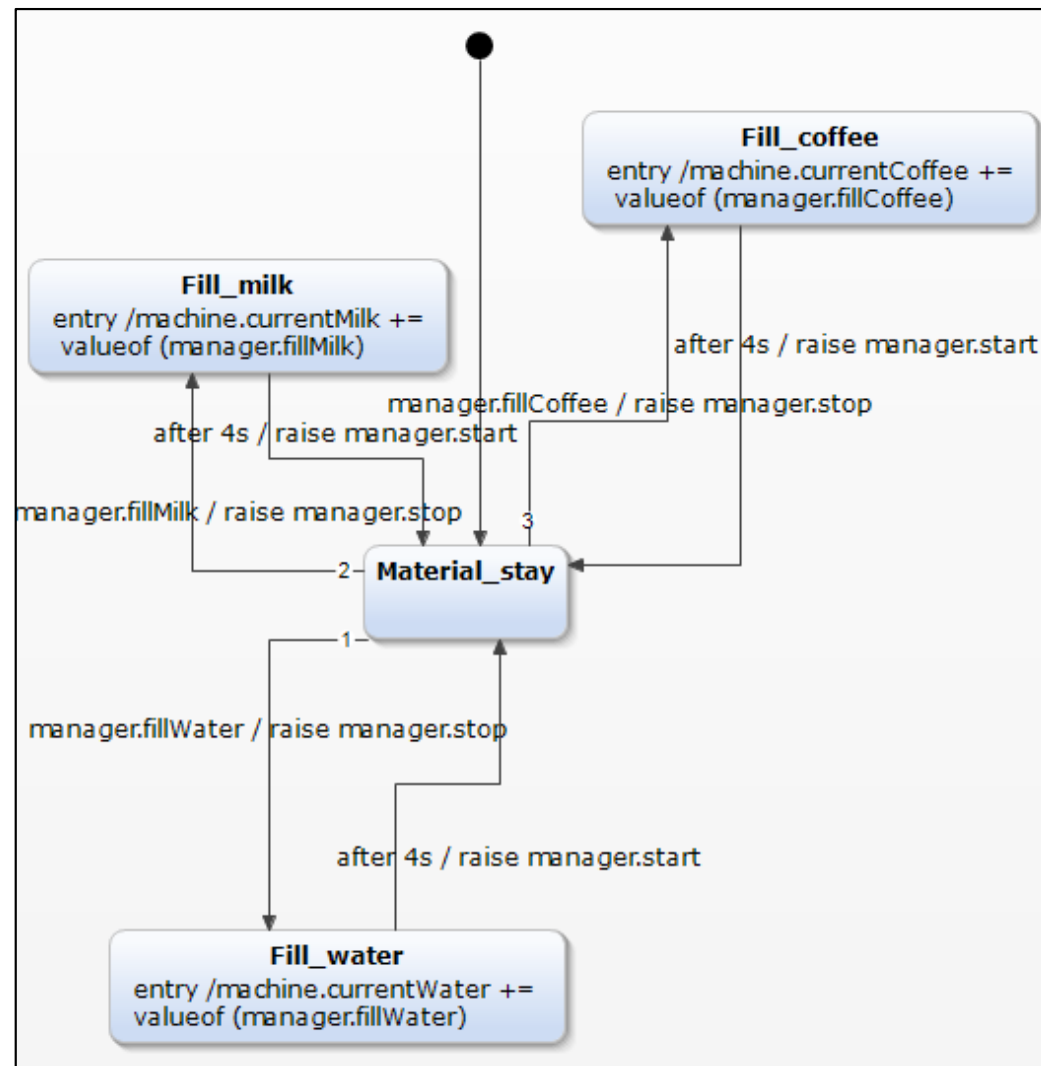
On_operation state - Operation_region

- An example of the specialMilk



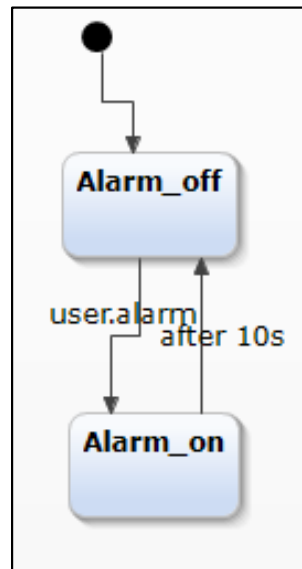
On_operation state - Material_region

- 재료 보충 관련 region
 - Coin_region을 stop으로 변경 후 재료 보충 진행



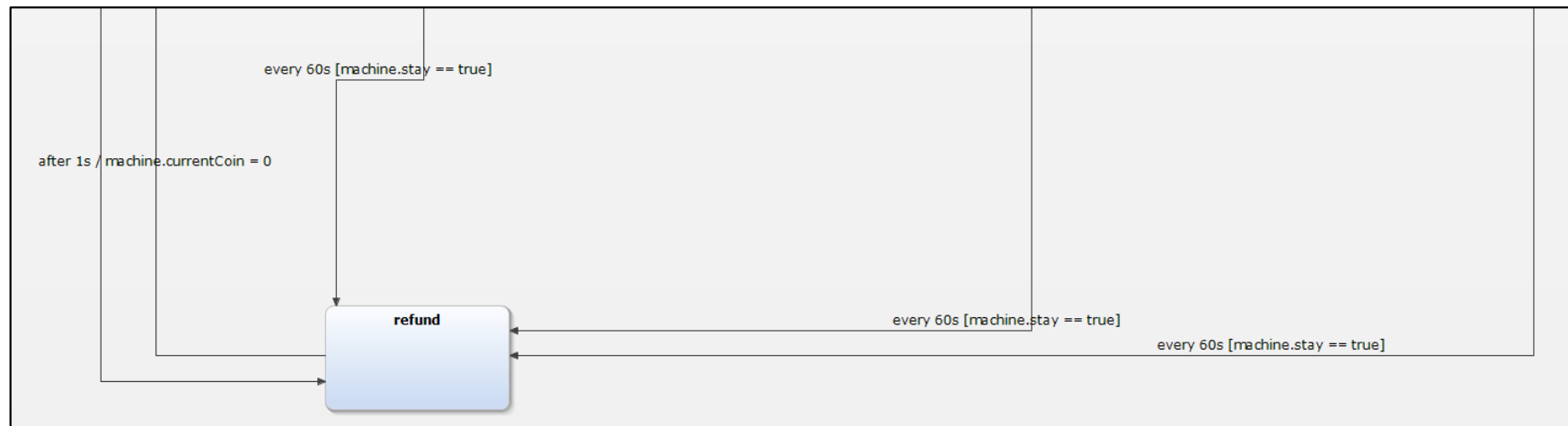
On_operation state - Alarm_region

- Alarm 관련 region
 - 알람 이벤트 후 일정 시간 후 off



Refund state

- 환불 event, 재료 부족 event에 따라 환불 진행하는 state



SIMULATION

Simulation

- Start simulation

The screenshot shows an IDE window titled 'Coffee_machine' with a statechart diagram. The statechart is a UML state machine diagram for a coffee machine. It starts with a state 'Coin_ready' which transitions to 'Operation_ready' upon the event 'Coin_ready'. From 'Operation_ready', there are three parallel regions for different coffee types: 'Special_Milk', 'Milk', and 'Special_black'. Each region contains a sequence of states: 'Prepare_coffee', 'Mix_water', and 'Mix_milk'. Transitions between these states are triggered by 'timer' events. The diagram also includes a 'refund' state and various 'end' and 'alarm' actions. A context menu is open over the statechart, with 'Run As' selected, and 'Statechart Simulation' highlighted. The Properties view at the bottom shows the statechart name as 'coffee_main' and the diagram as 'coffee_main'.

Simulation

- Insert the event and value in simulation proper

coffee_main [active] 00:00:07.600

Name	Value
start	
on	
off	
user	
insertCoin	0
refund	
specialMilk	
milk	
specialBlack	
alarm	
machine	
(x) currentCoin	0
(x) currentMilk	1000
(x) currentWater	1000
(x) currentCoffee	500
(x) stay	<input type="checkbox"/> false
manager	
fillWater	0
fillCoffee	0
fillMilk	0
stop	
start	
time events	
Coffee_specialMilk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_milk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_specialBlack_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
end_time_event_0	0
Fill_water_time_event_0	0
Fill_milk_time_event_0	0
Fill_coffee_time_event_0	0
Alarm_on_time_event_0	0
refund_time_event_0	0

Event with value

Simulation

ws - Coffee_machine/model/coffee_main.sct - YAKINDU SCT

File Edit Diagram Navigate Search Project Run Window Help

Project Explorer coffee_main

Statechart coffee_main

```

@CycleBased(200)

Interface start:
in event on
in event off

Interface user:
in event insertCoin : integer
in event refund
in event specialMilk
in event milk
in event specialBlack
in event alarm

Interface machine:
var currentCoin : integer
var currentMilk : integer = 1000
var currentWater : integer = 1000
var currentCoffee : integer = 500
var stay : boolean = false

Interface manager:
in event fillWater : integer
in event fillCoffee : integer
in event fillMilk : integer
in event stop
in event start

// Define events and
// and variables here
// Use CTRL + Space for content assist b
    
```

State representation in simulation

Simulation 00:00:54.800

Name	Value
start	F on
off	F off
user	
insertCoin	0
refund	F refund
specialMilk	F specialMilk
milk	F milk
specialBlack	F specialBlack
alarm	F alarm
machine	
currentCoin	1000
currentMilk	1000
currentWater	1000
currentCoffee	500
stay	<input type="checkbox"/> false
manager	
fillWater	0
fillCoffee	0
fillMilk	0
stop	0
start	0
time events	
Coffee_specialMilk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_milk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_specialBlack_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
end_time_event_0	0
fill_water_time_event_0	0
fill_milk_time_event_0	0
fill_coffee_time_event_0	0
Alarm_on_time_event_0	0
refund_time_event_0	0

Properties Statechart coffee_main

Model Statechart Name: coffee_main

Diagram Statechart domain

Appearance Default

Statechart Behavior: @CycleBased(200)

Documentation:

Region Priority: Region Main region

Simulation [1/4]

Case of special milk

The screenshot shows a state machine simulation environment. The main window displays a statechart for a coffee machine. The statechart is divided into several states, including 'idle', 'fillingWater', 'fillingMilk', 'mixing', 'brewing', and 'dispensing'. A 'specialMilk' state is highlighted in yellow. The left pane shows the code for the state machine, including interfaces for start, user, machine, and manager. The right pane shows a simulation log with a table of variables and their values.

Simulation Log Table:

Name	Value
start	
on	
off	
user	
insertCoin	0
refund	
specialMilk	
milk	
specialBlack	
alarm	
machine	
currentCoin	800
currentMilk	960
currentWater	960
currentCoffee	420
stay	<input type="checkbox"/> false
manager	
fillWater	0
fillCoffee	0
fillMilk	0
stop	
start	
time events	
Coffee_specialMilk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_milk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_specialBlack_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
end_time_event_0	0
fill_water_time_event_0	0
fill_milk_time_event_0	0
fill_coffee_time_event_0	0
Alarm_on_time_event_0	0
refund_time_event_0	0

Simulation [3/4]

The screenshot displays a UML modeling environment with a statechart for a coffee machine. The statechart is divided into several states, including 'idle', 'fillingWater', 'fillingMilk', 'mixing', and 'dispensing'. A callout box highlights a transition labeled 'Decrease water'. The right sidebar shows a simulation table with the following data:

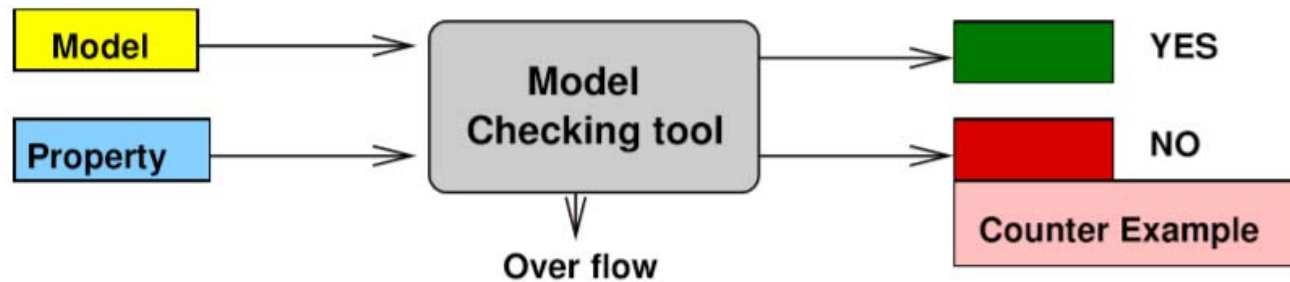
Name	Value
start	
on	
off	
user	
insertCoin	0
refund	
specialMilk	
milk	
specialBlack	
alarm	
machine	
currentCoin	800
currentMilk	960
currentWater	950
currentCoffee	400
stay	false
manager	

The bottom right sidebar shows a list of time events:

time events	Value
Coffee_specialMilk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_milk_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
Mix_milk_time_event_0	0
end_time_event_0	0
Coffee_specialBlack_time_event_0	0
Prepare_coffee_time_event_0	0
Mix_water_time_event_0	0
end_time_event_0	0
Fill_water_time_event_0	0
Fill_milk_time_event_0	0
Fill_coffee_time_event_0	0
Alarm_on_time_event_0	0
refund_time_event_0	0

Introduction to SMV

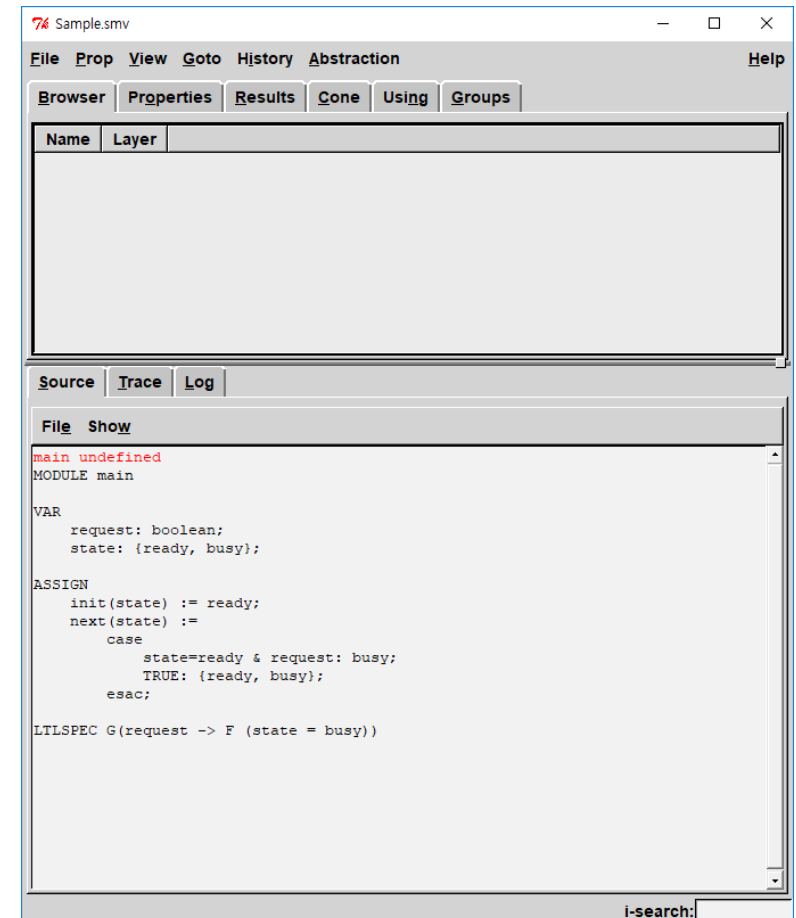
Model Checking



- Model checking
 - An automatic technique for verifying properties of a finite model of a system.
- General approach:
 - Construct M ← a model of the behavior of the system
(given as kripke structure, finite automata). M must be finite.
 - Specify ϕ ← a property expected of the system (given as Temporal Logic)
 - Check that M satisfies ϕ , if not , produce counter-example.
- Examples of model checking tools:
 - SMV, SPIN, UPPAAL, Kronos

SMV: Symbolic Model Verifier

- Ken McMillan
 - **Symbolic Model Checking: An Approach to State Explosion Problem, 1993**
- Modeling Language
 - Modularized and hierarchical descriptions
 - Finite data types: boolean, enum, int, etc.
 - Array, loops, if-close, etc.
 - Non-determinism, parallel execution
- Property specification Language
 - CTL and LTL
 - safety, liveness, deadlock
 - Fairness
- **Cadence SMV**: command line and GUI for Windows / Linux / Sun-OS
 - Other SMV versions: CMU-SMV, **NuSMV**

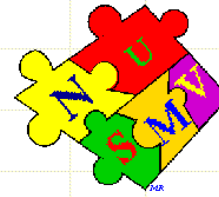


```

7% Sample.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
Name Layer
Source Trace Log
File Show
main undefined
MODULE main
VAR
  request: boolean;
  state: {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state=ready & request: busy;
      TRUE: {ready, busy};
    esac;
LTLSPEC G(request -> F (state = busy))
i-search:
  
```

NuSMV

- Re-implementation at IRST
- <http://nusmv.fbk.eu/>



NuSMV: a new symbolic model checker

[New! NuSMV 2.6.0 is OUT!](#)

[New! nuXmv 1.0.0](#) a new symbolic model checker for the analysis of synchronous finite-state and infinite-state systems is OUT

NuSMV 2.6.0 is a major release that comes after four years passed working under the surface. The release provides some new features, many bug fixes and optimizations, and substantial differences in the software architecture and building system.

Follow this [link](#) to retrieve a copy.

Read the [announce for NuSMV 2.6.0](#).

Read the [announce for NuSMV 2.5.4](#).

Read the [announce for NuSMV 2.5.3](#).

Read the [announce for NuSMV 2.5.2](#).

Read the [announce for NuSMV 2.5.1](#).

Read the [announce for NuSMV 2.5.0](#).

Read the [announce for NuSMV 2.4.3](#).

Read the [announce for NuSMV 2.4.2](#).

Read the [announce for NuSMV 2.4.1](#).

Read the [announce for NuSMV 2.4.0](#).

Read the [announce for NuSMV 2.3.1](#).

Read the [announce for NuSMV 2.3.0](#).

Read the [announce for NuSMV 2.2.5](#).

Read the [announce for NuSMV 2.2.4](#).

Read the [announce for NuSMV 2.2.3](#).

Read the [announce for NuSMV 2.2.2](#).

Read the [announce for NuSMV 2.2.1](#).

Read the [announce for NuSMV 2.2](#).

Read the [announce for NuSMV 2.1](#).

Read the [announce for NuSMV 2](#).

[Links to some projects using NuSMV](#)

[NuSMV 2 is OpenSource!](#)

New versions of NuSMV are distributed under the [LGPL v2.1](#) license. This is an Open Source license that allows free academic and commercial usage of NuSMV. For further information follow [this link](#).



[NuSMV](#) is a symbolic model checker developed as a joint project between:

Running NuSMV (Interactively)

- **NuSMV -int**
 - Runs NuSMV in interactive mode
- **read_model -i <filename>**
 - Reads a system spec. from file
- **go**
 - Builds the internal representation of the model
- **check_fsm**
 - Checks whet
- **compute_reachable**
 - Computes set of reachable states first
 - The model checking algorithm traverses only the set of reachable states instead of complete state space.
 - Useful if reachable state space is a small fraction of total state space
- **check_ctlspec [check_ltlspec]**
 - Checks all the CTL properties [LTL properties] included in the file

A Sample SMV Program

MODULE main

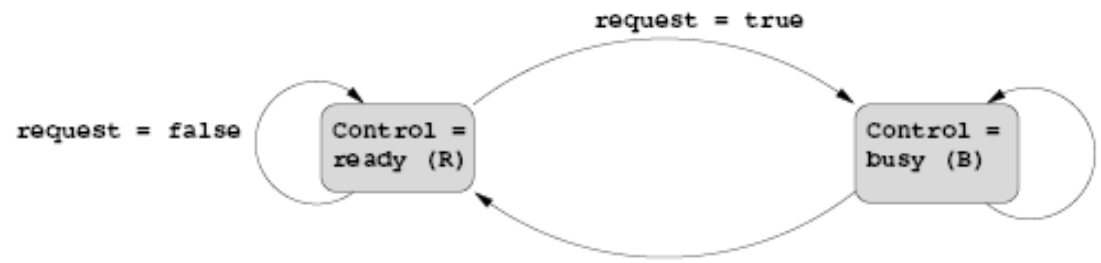
VAR

request: boolean;
state: {ready, busy};

ASSIGN

init(state) := ready;
next(state) :=
 case
 state=ready & request: busy;
 1: {ready, busy};
 esac;

LTLSPEC G(request -> F (state = busy))



NuSMV

NuSMV provides:

1. A language for describing finite state models of systems
 - ▶ Reasonably expressive
 - ▶ Allows for modular construction of models
2. Model checking algorithms for checking specifications written in LTL and CTL (and some other logics) against finite state machines.

A first SMV program

```
MODULE main
  VAR
    b0 : boolean
  ASSIGN
    init(b0) := FALSE;
    next(b0) := !b0;
```

An SMV program consists of:

- ▶ Declarations of state variables (b0 in the example); these determine the state space of the model.
- ▶ Assignments that constrain the valid initial states (`init(b0) := FALSE`).
- ▶ Assignments that constrain the transition relation (`next(b0) := !b0`).

Declaring state variables

SMV data types include:

boolean:

```
x : boolean;
```

enumeration:

```
st : {ready, busy, waiting, stopped};
```

bounded integers (intervals):

```
n : 1..8;
```

arrays and bit-vectors

```
arr : array 0..3 of {red, green, blue};
```

```
bv : signed word[8];
```

Assignments

initialisation:

ASSIGN

init(x) := expression ;

progression:

ASSIGN

next(x) := expression ;

immediate:

ASSIGN

y := expression ;

or

DEFINE

y := expression ;

Assignments

- ▶ If no **init()** assignment is specified for a variable, then it is initialised non-deterministically;
- ▶ If no **next()** assignment is specified, then it evolves nondeterministically. i.e. it is unconstrained.
 - ▶ Unconstrained variables can be used to model nondeterministic inputs to the system.
- ▶ Immediate assignments constrain the current value of a variable in terms of the current values of other variables.
 - ▶ Immediate assignments can be used to model outputs of the system.

Expressions

$expr$::=	atom	symbolic constant
		number	numeric constant
		id	variable identifier
		$! expr$	logical not
		$expr \bowtie expr$	binary operation
		$expr[expr]$	array lookup
		$next(expr)$	next value
		$case_expr$	
		set_expr	

where $\bowtie \in \{\&, |, +, -, *, /, =, !=, <, <=, \dots\}$

Case Expression

```
case_expr ::=  
  case  
    expra1 : exprb1;  
    ...  
    expran : exprbn;  
  esac
```

- ▶ Guards are evaluated sequentially.
- ▶ The first true guard determines the resulting value

Set expressions

Expressions in SMV do not necessarily evaluate to one value.

- ▶ In general, they can represent a set of possible values.
`init(var) := {a,b,c} union {x,y,z} ;`
- ▶ destination (lhs) can take any value in the set represented by the set expression (rhs)
- ▶ constant `c` is a syntactic abbreviation for singleton `{c}`

LTL Specifications

- ▶ LTL properties are specified with the keyword LTLSPEC:
LTLSPEC <ltl_expression> ;
- ▶ <ltl_expression> can contain the temporal operators:
X_ F_ G_ _U_
- ▶ E.g. condition `out = 0` holds until `reset` becomes false:
LTLSPEC (`out = 0`) U (!reset)

ATM Example

```
MODULE main
```

```
VAR
```

```
  state: {welcome, enterPin, tryAgain, askAmount,  
         thanksGoodbye, sorry};
```

```
  action: {cardIn, correctPin, wrongPin, ack, cancel,  
         fundsOK, problem, none};
```

```
ASSIGN
```

```
  init(state) := welcome;
```

```
  next(state) := case
```

```
    state = welcome & action = cardIn      : enterPin;
```

```
    state = enterPin & action = correctPin  : askAmount ;
```

```
    state = enterPin & action = wrongPin    : tryAgain;
```

```
    state = tryAgain & action = ack         : enterPin;
```

```
    state = askAmount & action = fundsOK   : thanksGoodbye;
```

```
    state = askAmount & action = problem   : sorry;
```

```
    state = enterPin & action = cancel      : thanksGoodbye;
```

```
    TRUE                                   : state;
```

```
  esac;
```

```
LTLSPEC F( G state = thanksGoodbye  
          | G state = sorry  
          );
```

Model을 수정해서 해당
Spec.을 만족하게끔 합니다!

Running NuSMV

Batch

```
$ NuSMV atm.smv
```

Interactive

```
$ NuSMV -int atm.smv  
NuSMV > go  
NuSMV > check_ltlspec  
NuSMV > quit
```

- ▶ go abbreviates the sequence of commands `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`.
- ▶ For command options, use `-h` or look in the NuSMV User Manual.

Expected Failure

```
NuSMV > check_ltlspec
-- specification  F ( G state = thanksGoodbye
                   |  G state = sorry)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
    state = welcome
    input = cardIn
-> State: 1.2 <-
    state = enterPin
    input = correctPin
-- Loop starts here
-> State: 1.3 <-
    state = askAmount
    input = ack
-> State: 1.4 <-
```


Unexpected Failure

```
-- specification
  ( F ( G !(state = askAmount)) ->
    F ( G state = thanksGoodbye | G state = sorry))
    is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  state = welcome
  input = cardIn
-- Loop starts here
-> State: 2.2 <-
  state = enterPin
  input = ack
-> State: 2.3 <-
```

Success

```
-- specification
  ( G (((state = welcome -> F input = cardIn) &
        (state = enterPin ->
          F (state = enterPin &
            (input = correctPin | input = cancel)))) &
        (state = askAmount -> F (input = fundsOK
                                  | input = problem))) ->
    F ( G state = thanksGoodbye | G state = sorry))
is true
```

```
C:\Users\JUNBEOM YOO>NuSMV ATM.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification F ( G state = thanksGoodbye | G state = sorry) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = welcome
  action = cardIn
-> State: 1.2 <-
  state = enterPin
  action = correctPin
-- Loop starts here
-> State: 1.3 <-
  state = askAmount
  action = ack
-> State: 1.4 <-
-- specification ( F ( G !(state = askAmount)) -> F ( G state = thanksGoodbye | G state = sorry)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  state = welcome
  action = cardIn
-- Loop starts here
-> State: 2.2 <-
  state = enterPin
  action = ack
-> State: 2.3 <-
-- specification ( G (((state = welcome -> F action = cardIn) & (state = enterPin -> F (state = enterPin & (action = correctPin | action =
cancel)))) & (state = askAmount -> F (action = fundsOK | action = problem))) -> F ( G state = thanksGoodbye | G state = sorry)) is true

C:\Users\JUNBEOM YOO>
```

Modules

```
MODULE counter
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := (digit + 1) mod 10;
```

```
MODULE main
VAR c0 : counter;
    c1 : counter;
    sum : 0..99;
ASSIGN
  sum := c0.digit + 10 * c1.digit;
```

- ▶ Modules are instantiated in other modules. The instantiation is performed inside the VAR declaration of the parent module.
- ▶ In each SMV specification there must be a module main. It is the top-most module.
- ▶ All the variables declared in a module instance are visible in the module in which it has been instantiated via the dot notation (e.g., `c0.digit`, `c1.digit`).

Modules

```
MODULE counter
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := (digit + 1) mod 10;
```

```
MODULE main
VAR c0 : counter;
    c1 : counter;
    sum : 0..99;
ASSIGN
  sum := c0.digit + 10 * c1.digit;
```

```
LTLSPEC
  F sum = 13;
```

- ▶ Is this specification satisfied by this model?

```
-- specification F sum = 13 is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  c0.digit = 0
  c1.digit = 0
  sum = 0
-> State: 1.2 <-
  c0.digit = 1
  c1.digit = 1
  sum = 11
-> State: 1.3 <-
  c0.digit = 2
  c1.digit = 2
  sum = 22
...
```

```

C:\WINDOWS\system32\cmd.exe
C:\Users\#JUNBEOM YOO>NuSMV Module1.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification F sum = 13 is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  c0.digit = 0
  c1.digit = 0
  sum = 0
-> State: 1.2 <-
  c0.digit = 1
  c1.digit = 1
  sum = 11
-> State: 1.3 <-
  c0.digit = 2
  c1.digit = 2
  sum = 22
-> State: 1.4 <-
  c0.digit = 3
  c1.digit = 3
  sum = 33
-> State: 1.5 <-
  c0.digit = 4
  c1.digit = 4
  sum = 44
-> State: 1.6 <-
  c0.digit = 5
  c1.digit = 5
  sum = 55
-> State: 1.7 <-
  c0.digit = 6
  c1.digit = 6
  sum = 66
-> State: 1.8 <-
  c0.digit = 7
  c1.digit = 7
  sum = 77
-> State: 1.9 <-
  c0.digit = 8
  c1.digit = 8
  sum = 88
-> State: 1.10 <-
  c0.digit = 9
  c1.digit = 9
  sum = 99
-> State: 1.11 <-
  c0.digit = 0
  c1.digit = 0
  sum = 0
C:\Users\#JUNBEOM YOO>_

```

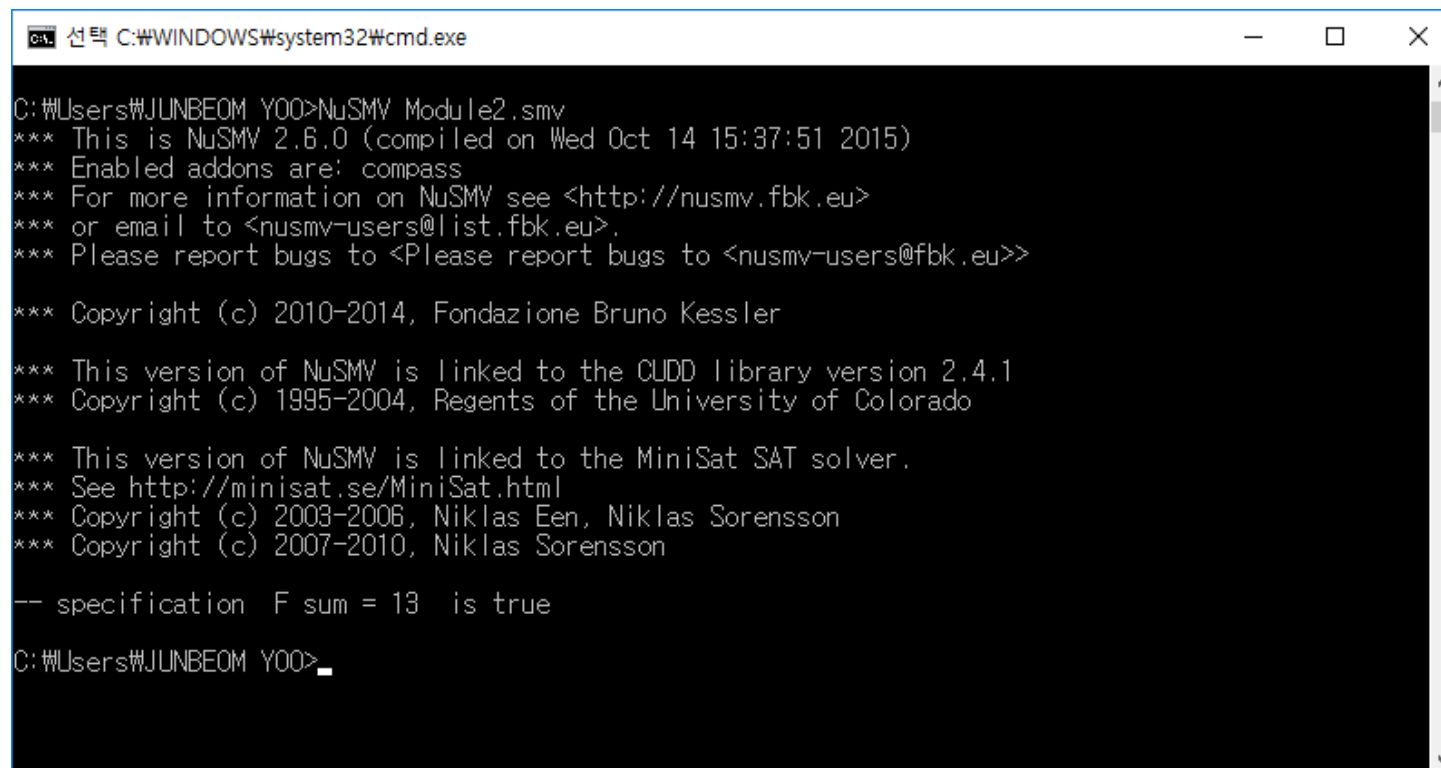
Modules with parameters

```
MODULE counter(inc)
VAR digit : 0..9;
ASSIGN
  init(digit) := 0;
  next(digit) := inc ? (digit + 1) mod 10
                : digit;
DEFINE top := digit = 9;

MODULE main
VAR c0 : counter(TRUE);
    c1 : counter(c0.top);
    sum : 0..99;
ASSIGN
  sum := c0.digit + 10 * c1.digit;
```

- ▶ Formal parameters (inc) are substituted with the actual parameters (TRUE, c0.top) when the module is instantiated.
- ▶ Actual parameters can be any legal expression.
- ▶ Actual parameters are passed by reference.


```
-- specification F sum = 13 is true
```



```
선택 C:\WINDOWS\system32\cmd.exe
C:\Users\JUNBEOM YOO>NuSMV Module2.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification F sum = 13 is true
C:\Users\JUNBEOM YOO>_
```

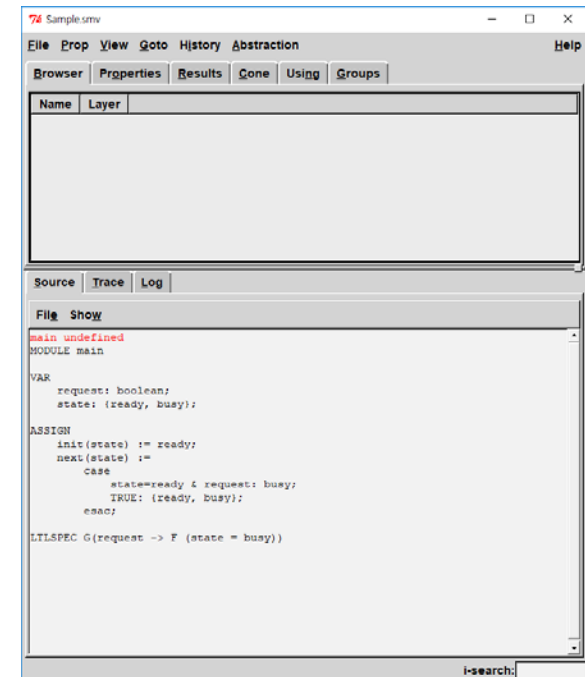
Summary

- ▶ Introduction to NuSMV
 - ▶ H&R Section 3.3
 - ▶ NuSMV Tutorial:
<http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf>
 - ▶ NuSMV Start-up Guide on FV Web Page
- ▶ Next time:
 - ▶ Introduction to the practical exercise.

NuSMV Verification

The NuSMV Verification - CVM

- Let's perform the NuSMV verification upon CVM (Coffee Vending Machine).
- Modeling :
 - The SMV input program (.SMV)
 - Use your Statecharts model as a base reference.
- Formal Verification :
 - Use NuSMV or Cadence SMV
- Properties to verify
 - Deadlock freeness
 - Basic functions
 - Important functions
 - “100원을 넣고 커피를 누르면, 항상 커피가 나온다.”
 - “동전을 넣지 않으면, 커피가 절대로 나오지 않는다.”
 - “100원을 넣고 커피3을 누르면, 반드시 커피3이 나온다.”
 - “환불을 누르면, 남은 금액이 환불된다.”
 - “재고가 없을 때는 절대로 커피가 나오지 않는다.”



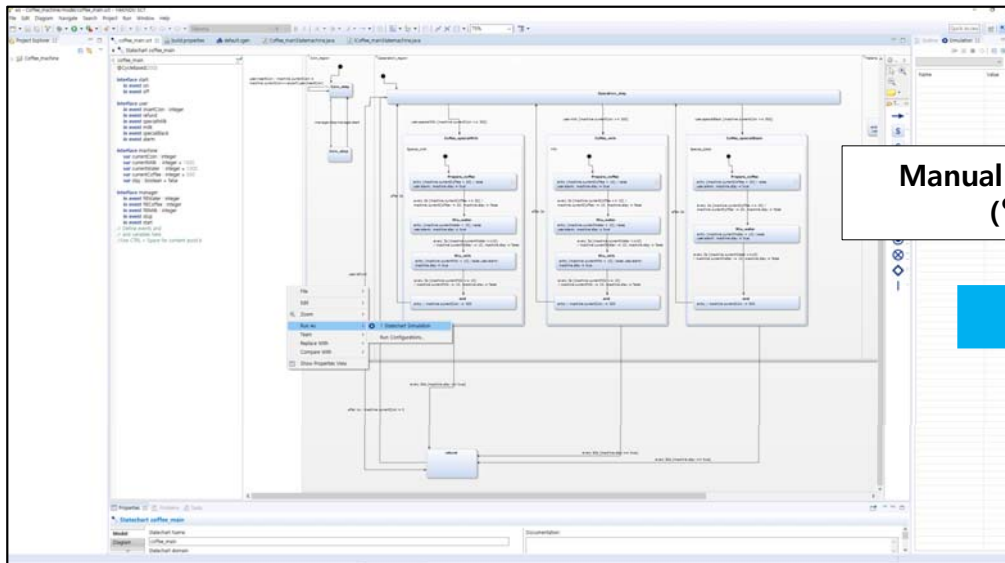
```

Sample.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
Name Layer
Source Trace Log
File Show
main undefined
MODULE main
VAR
  request: boolean;
  state: (ready, busy);
ASSIGN
  init(state) := ready;
  next(state) :=
    CASE
      state=ready & request: busy;
      TRUE: (ready, busy);
    esac;
LTLSPEC G(request -> F (state = busy))
  
```

The Cadence SMV

Verification with symbolic model checker

- NuSMV
 - [NuSMV](#) is a reimplementation and extension of [SMV](#), the first model checker based on BDDs. [NuSMV](#) has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas.



Manual Transformation
(일부 축소)



```

1  MODULE main
2
3  VAR
4    state : {off, On_operation, refund};
5    onoff_command : boolean;
6    refund_command : boolean;
7    current_coin : 0..2000;
8
9    alarm_timeout : process timer();
10   check : process coin(state, current_coin);
11   on : process alarm(state, alarm_timeout);
12   e : {specialMilk, milk, black};
13
14
15  init(state) := off;
16  next(state) :=
17    case
18      state = off & onoff_command = TRUE : On_operation;
19      state = On_operation & onoff_command = FALSE : off;
20      state = On_operation & refund_command = TRUE : refund;
21      state = refund : On_operation;
22      TRUE : state;
23    esac;
24
25  next(current_coin) :=
26    case
27      state = off | state = On_operation : current_coin;
28      state = refund : 0;
29    esac;
30
31  SPEC EF (state = off -> state = On_operation)
32  SPEC EF (state = On_operation -> current_coin != 0)
33  SPEC AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE)
34  SPEC AX (state = refund -> current_coin = 0)
35  -----
36
37  MODULE coin(machinestate, current_coin)
38  VAR

```

MODULE coin

- Coin 의 증가/감소에 대한 모듈
 - 증가: 0, 100, 500, 1000 중 선택하여 증가
 - 감소: 추출 시 커피의 가격에 맞게 감소

```

MODULE coin(machinestate, current_coin2, coffee_command)
VAR
  coin_value : {0, 100, 500, 1000};
  coin_reduce : {0, 200, 300};

ASSIGN
  init(coin_value) := 0;
  next(coin_value) :=
    case
      machinestate = On_operation : {0, 100, 500, 1000};
      TRUE : 0;
    esac;

  init(coin_reduce) := 0;
  next(coin_reduce) :=
    case
      machinestate = On_operation & coffee_command = specialMilk: 300;
      machinestate = On_operation & coffee_command = milk: 200;
      machinestate = On_operation & coffee_command = black: 300;
      TRUE : 0;
    esac;

DEFINE
  main.current_coin := current_coin2 + coin_value - coin_reduce;

```


MODULE timer

- Alarm의 timer
 - 1000 cycle 후 종료 하도록 timeout 전달

```

MODULE timer
VAR
  time : 0..1000;
ASSIGN
  init(time) := 0;
  next(time) := (time + 1) mod 1000;
DEFINE
  timeout_alarm := time mod 1000 = 0;

```

MODULE alarm

- 재료 부족 시 알람에 대한 모듈
 - 재료 부족이 TRUE일 경우 alarm on
 - 일정 시간 이후 (timeout) off로 전환

```

MODULE alarm(material_lack, timeout)
VAR
    alarm_on : boolean;

ASSIGN
    init(alarm_on) := FALSE;
    next(alarm_on) :=
        case
            material_lack = TRUE : TRUE;
            timeout.timeout_alarm = TRUE : FALSE;
            TRUE : FALSE;
        esac;

```

MODULE main

- Vending machine의 main
 - 현재 머신의 상태 관리 (off, on_operation, refund)
 - 현재 머신의 정보 저장 (water, milk, coffee, coin)
 - 다른 MODULE process (timer, alarm, coin)
 - Command에 따라 상태 변화 (onoff, refund, coffee)
 - 커피 추출에 따른 상태 변화 및 정보 변경

MODULE main [VAR]

- FSM을 구성하는 모든 요소를 Variables로 표현합니다.
 - External Events / Internal Events
 - State Variables / States
- 각 변수에 대한 ASSIGN을 정의합니다.
 - Assign이 없는 변수는 random하게 (nondeterministic) 값이 할당됩니다.

VAR

```

state : {off, On_operation, refund};
onoff_command : boolean;
refund_command : boolean;
material_lack : boolean;
coffee_command : {NONE, specialMilk, milk, black};

alarm_timeout : process timer();
coin_check : process coin(state, current_coin, coffee_command);
alarm_on : process alarm(material_lack, alarm_timeout);

current_water: 0..1000;
current_milk: 0..1000;
current_coffee: 0..1000;
current_coin : 0..2000;

```

MODULE main [ASSIGN]

```

ASSIGN
init(state) := off;|
next(state) :=
  case
  state = off & onoff_command = TRUE : On_operation;
  state = On_operation & onoff_command = FALSE : off;
  state = On_operation & refund_command = TRUE : refund;
  state = refund : On_operation;
  TRUE : state;
  esac;

init(current_milk) := 1000;
next(current_milk) :=
  case
  current_coin >= 300 & state = On_operation & coffee_command = specialMilk & current_milk >= 20 : current_milk - 20;
  current_coin >= 200 & state = On_operation & coffee_command = milk & current_milk >= 10 : current_milk - 10;
  state = On_operation & coffee_command = specialMilk & current_milk < 20 : current_milk;
  TRUE : current_milk;
  esac;

init(current_water) := 1000;
next(current_water) :=
  case
  current_coin >= 300 & state = On_operation & coffee_command = specialMilk & current_water >= 20 : current_water - 20;
  current_coin >= 300 & state = On_operation & coffee_command = black & current_water >= 20 : current_water - 20;
  current_coin >= 200 & state = On_operation & coffee_command = milk & current_water >= 20 : current_water - 20;
  TRUE : current_water;
  esac;

init(current_coffee) := 1000;
next(current_coffee) :=
  case
  current_coin >= 300 & state = On_operation & coffee_command = specialMilk & current_coffee >= 20 : current_coffee - 20;
  current_coin >= 300 & state = On_operation & coffee_command = black & current_coffee >= 30 : current_coffee - 30;
  current_coin >= 200 & state = On_operation & coffee_command = milk & current_coffee >= 10 : current_coffee - 10;
  TRUE : current_coffee;
  esac;

```

MODULE main [ASSIGN]

```

init(coffee_command) := NONE;
next(coffee_command) :=
  case
  state = On_operation : {NONE, specialMilk, milk, black};
  state != On_operation : NONE;
  esac;

init(material_lack) := FALSE;
next(material_lack) :=
  case
  current_milk < 10 | current_coffee < 10 | current_water < 20 : TRUE;
  coffee_command = black & current_coffee < 30 : TRUE;
  coffee_command = specialMilk & (current_milk < 20 | current_coffee < 20) : TRUE;
  TRUE : FALSE;
  esac;

init(current_coin) := 0;
next(current_coin) :=
  case
  state = off | state = On_operation : current_coin;
  state = refund : 0;
  esac;

```

MODULE main [SPEC]

```

SPEC AG EX TRUE

SPEC EF (state = off -> state = On_operation)
SPEC EF (state = On_operation -> state = off)

SPEC EF (state = On_operation -> current_coin != 0)
SPEC AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE)
SPEC AX (state = refund & current_coin != 0 -> current_coin = 0)

SPEC AX (current_coin = 300 & state = On_operation & coffee_command = specialMilk -> (current_coin = 0))
SPEC AX (current_coin = 500 & state = On_operation & coffee_command = black -> (current_coin = 200))

SPEC AF (current_milk = 1000 & state = On_operation & coffee_command = specialMilk -> EX(current_milk = 980))
SPEC AF (current_coffee = 1000 & state = On_operation & coffee_command = black -> EX(current_coffee = 970))
SPEC AG (current_coffee < 30 & state = On_operation & coffee_command = black -> alarm_on.alarm_on = TRUE)
SPEC AG (current_milk < 20 & state = On_operation & coffee_command = specialMilk -> alarm_on.alarm_on = TRUE)

```

```

-- specification AG (EX TRUE) is true
-- specification EF (state = off -> state = On_operation) is true
-- specification EF (state = On_operation -> state = off) is true
-- specification EF (state = On_operation -> current_coin != 0) is true
-- specification AX ((state = refund & current_coin != 0) -> current_coin = 0) is true
-- specification AX (((current_coin = 300 & state = On_operation) & coffee_command = specialMilk) -> current_coin = 0) is true
-- specification AX (((current_coin = 500 & state = On_operation) & coffee_command = black) -> current_coin = 200) is true
-- specification AF (((current_milk = 1000 & state = On_operation) & coffee_command = specialMilk) -> EX current_milk = 980) is true
-- specification AF (((current_coffee = 1000 & state = On_operation) & coffee_command = black) -> EX current_coffee = 970) is true
-- specification AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE) is true
-- specification AG (((current_coffee < 30 & state = On_operation) & coffee_command = black) -> alarm_on.alarm_on = TRUE) is true
-- specification AG (((current_milk < 20 & state = On_operation) & coffee_command = specialMilk) -> alarm_on.alarm_on = TRUE) is true

```

CTL property

No.	CTL SPEC	Description
1	SPEC AG EX TRUE	Deadlock
2	SPEC EF (state = off -> state = On_operation)	머신이 off 상태 일 때 on 이 될 수 있다.
3	SPEC EF (state = On_operation -> state = off)	머신이 on 상태 일 때 off 될 수 있다.
4	SPEC EF (state = On_operation -> current_coin != 0)	머신이 on 상태일 때 코인이 증가할 수 있다.
5	SPEC AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE)	알람이 on 이면 항상 미래에 off가 된다.
6	SPEC AX (state = refund & current_coin != 0 -> current_coin = 0)	Refund state 이면 항상 다음에 coin이 초기화 된다.
7	SPEC AX (current_coin = 300 & state = On_operation & coffee_command = specialMilk -> (current_coin = 0))	300 coin을 넣고 specialMilk를 요청하면 coin이 0 이 된다.
8	SPEC AX (current_coin = 500 & state = On_operation & coffee_command = black -> (current_coin = 200))	500 coin을 넣고 black을 요청하면 coin이 200 이 된다.
9	SPEC AF (current_milk = 1000 & state = On_operation & coffee_command = specialMilk -> EX(current_milk = 980))	specialMilk를 요청하면 우유가 20 감소한다.
10	SPEC AF (current_coffee = 1000 & state = On_operation & coffee_command = black -> EX(current_coffee = 970))	Black을 요청하면 커피가 30 감소한다.
11	SPEC AG (current_coffee < 30 & state = On_operation & coffee_command = black -> alarm_on.alarm_on = TRUE)	커피가 30 미만일 때 black을 요청하면 항상 알람이 울린다.
12	SPEC AG (current_milk < 20 & state = On_operation & coffee_command = specialMilk -> alarm_on.alarm_on =TRUE)	우유가 20 미만일 때 specialMilk를 요청하면 알람이 울린다.

The SMV Modeling Checking - Commands

```

선택 C:#WINDOWS#system32#cmd.exe - NuSMV -int
C:#Users#JUNBEOM YOO>NuSMV -int
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > read_model -i coffee2(jbyoo).smv
NuSMV > go
WARNING *** Processes are still supported, but deprecated. ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSES or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
NuSMV > check_fsm

#####
The transition relation is total: No deadlock state exists
#####
NuSMV > compute_reachable
The computation of reachable states has been completed.
The diameter of the FSM is 1004.

```

The SMV Modeling Checking – Verification Result

- 2 bugs seeded

```

NuSMV > check_ctlspec
-- specification AG (EX TRUE) is true
-- specification EF (state = off -> state = On_operation) is true
-- specification AG (state = off -> (state = On_operation & state = refund)) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = off
  onoff_command = FALSE
  refund_command = FALSE
  material_lack = FALSE
  coffee_command = NONE
  alarm_timeout.time = 0
  coin_check.coin_value = 0
  coin_check.coin_reduce = 0
  alarm_on.alarm_on = FALSE
  current_water = 1000
  current_milk = 1000
  current_coffee = 1000
  current_coin = 0
  alarm_timeout.timeout_alarm = TRUE
  coin_check.main.current_coin = 0
-- specification EF (state = On_operation -> state = off) is true
-- specification EF (state = On_operation -> current_coin != 0) is true
-- specification AX ((state = refund & current_coin != 0) -> current_coin = 0) is true
-- specification AX (((current_coin = 300 & state = On_operation) & coffee_command = specialMilk) -> current_coin = 0) is true
-- specification AX (((current_coin = 300 & state = On_operation) & coffee_command = specialMilk) -> current_coin = 100) is true
-- specification AX (((current_coin = 500 & state = On_operation) & coffee_command = black) -> current_coin = 200) is true
-- specification AF (((current_milk = 1000 & state = On_operation) & coffee_command = specialMilk) -> EX current_milk = 980) is true
-- specification AF (((current_coffee = 1000 & state = On_operation) & coffee_command = black) -> EX current_coffee = 970) is true
-- specification AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE) is true
-- specification AG (((current_coffee < 30 & state = On_operation) & coffee_command = black) -> alarm_on.alarm_on = TRUE) is true
-- specification AG (((current_milk < 20 & state = On_operation) & coffee_command = specialMilk) -> alarm_on.alarm_on = TRUE) is true
NuSMV >

```

The SMV Modeling Checking - Batch Mode

```

C:\WINDOWS\system32\cmd.exe
C:\Users\JUNBEOM YOO>NuSMV coffee2(jbyoo).smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:51 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

WARNING *** Processes are still supported, but deprecated. ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSES or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG (EX TRUE) is true
-- specification EF (state = off -> state = On_operation) is true
-- specification AG (state = off -> (state = On_operation & state = refund)) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = off
  onoff_command = FALSE
  refund_command = FALSE
  material_lack = FALSE
  coffee_command = NONE
  alarm_timeout.time = 0
  coin_check.coin_value = 0
  coin_check.coin_reduce = 0
  alarm_on.alarm_on = FALSE
  current_water = 1000
  current_milk = 1000
  current_coffee = 1000
  current_coin = 0
  alarm_timeout.timeout_alarm = TRUE
  coin_check.main.current_coin = 0
-- specification EF (state = On_operation -> state = off) is true
-- specification EF (state = On_operation -> current_coin != 0) is true
-- specification AX ((state = refund & current_coin != 0) -> current_coin = 0) is true
-- specification AX (((current_coin = 300 & state = On_operation) & coffee_command = specialMilk) -> current_coin = 0) is true
-- specification AX (((current_coin = 300 & state = On_operation) & coffee_command = specialMilk) -> current_coin = 100) is true
-- specification AX (((current_coin = 500 & state = On_operation) & coffee_command = black) -> current_coin = 200) is true
-- specification AF (((current_milk = 1000 & state = On_operation) & coffee_command = specialMilk) -> EX current_milk = 980) is true
-- specification AF (((current_coffee = 1000 & state = On_operation) & coffee_command = black) -> EX current_coffee = 970) is true
-- specification AF (alarm_on.alarm_on = TRUE -> alarm_on.alarm_on = FALSE) is true
-- specification AG (((current_coffee < 30 & state = On_operation) & coffee_command = black) -> alarm_on.alarm_on = TRUE) is true
-- specification AG (((current_milk < 20 & state = On_operation) & coffee_command = specialMilk) -> alarm_on.alarm_on = TRUE) is true
C:\Users\JUNBEOM YOO>

```