THE MODEL CHECKER SPIN

Shin Hong, KAIST

17th April, 2007

Contents

- SPIN is a software system to verify asynchronous process system model.

Introduction

- Software have taken big portion of system.
- These software should fulfill its responsibility.

Failure of System Software might be a disaster.

Introduction

- To avoid software system failure,
- We want to prove correctness of a system software.

→ **Verification of software** is needed.

Introduction

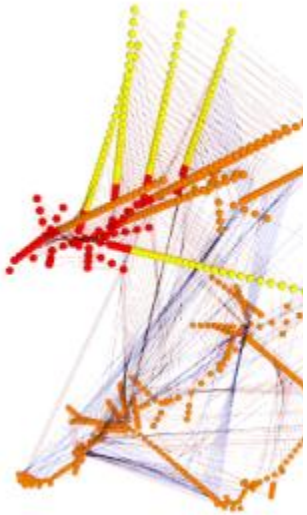
- Testing
  - Operate the program in some representative situation and verify whether it behaves as expected.
  - Dictum of Dijkstra

  " Program testing can be used to show the presence of bugs, but never to show their absence. "
- Model Checking
  - Describing a system as an FSM and verifying whether the system's behavior satisfies the desired properties.

Peterson's mutex

2 processes

3 processes

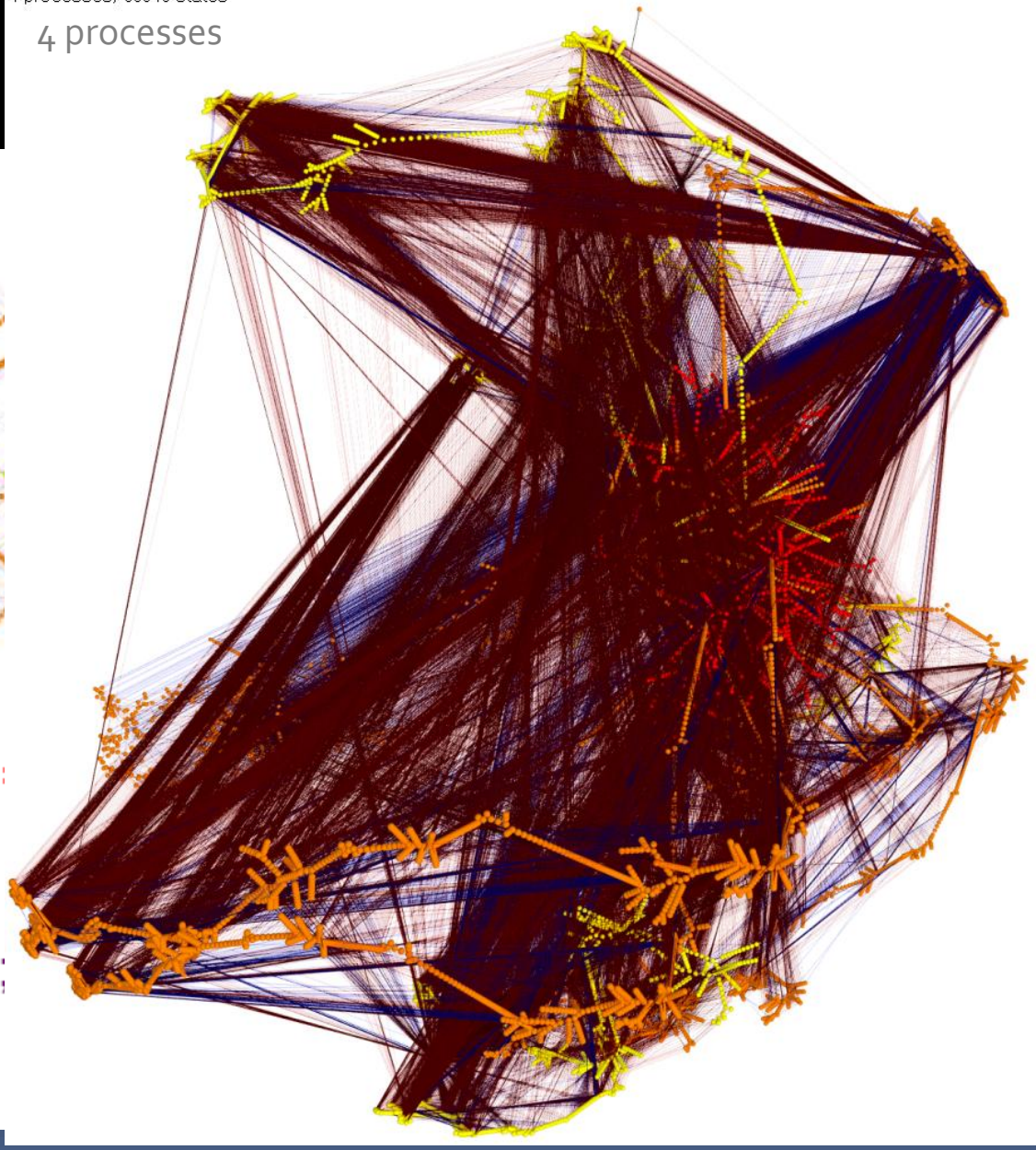


```
assert(ncrit =
ncrit--;

flag[_pid] = 0;
goto again;
}
```

4 processes, 55043 states:

4 processes



http://www.pst.ifi.lmu.de/~hammer/statespaces/peterson/index.html

Introduction

- We want to prove correctness of **system software** .
  - Concurrency

  → concurrent software may involve asynchrony.

  → Extremely complex.
  - Embedded System

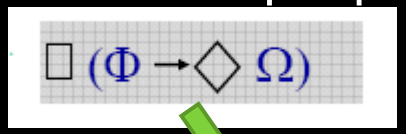
  → Completeness of verification is needed.

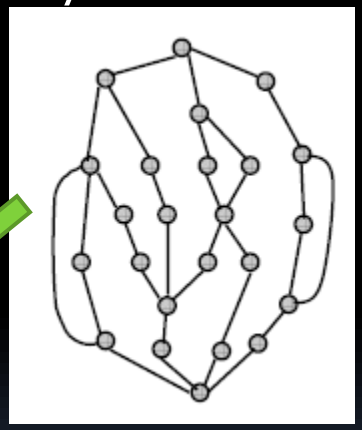
  →Model checking is more suitable for verification of system software.

Introduction

- Model Checking
  - Requirements
    ↓ modeling languages
  - Requirements properties

System
↓
System model





Model Checking
(State Exploration)

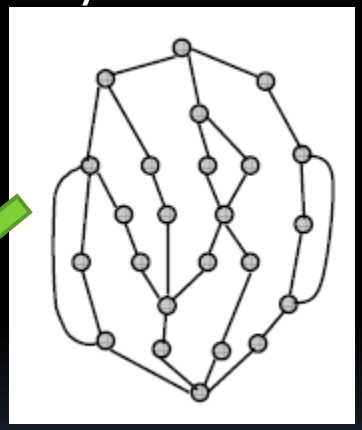
Satisfied → **Okay**

Not satisfied → **Counterexample**

Introduction

- Model Checking
  - Requirements ↓ Requirements properties
  - System ↓ System model

$$\Box(\Phi \rightarrow \Diamond \Omega)$$



Model Checking
(State Exploration)

Satisfied → **Okay**

Not satisfied → **Counterexample**

State Explosion

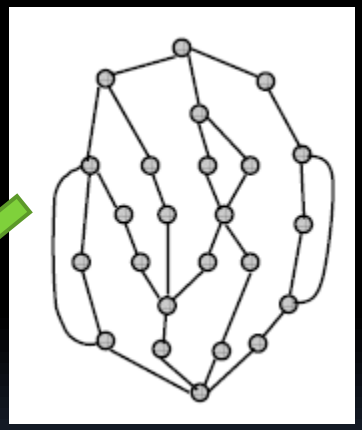
Introduction

- SPIN is designed to provide
  - An intuitive, program-like notation for specifying design choices unambiguously without implementation detail.
  - Powerful, concise, notation for expressing general correctness requirements.
  - Scalability : Reduce limitation of problem size, machine memory size, maximum runtime.

Introduction

- Model Checker SPIN
  - Requirements ↓ **Linear Temporal Logic**
  - System ↓ **PROMELA**
  - Requirements properties
  - System model

$$\Box(\Phi \rightarrow \Diamond \Omega)$$



On-the-fly verification
Negative Claim
Algorithms

Model Checking
(State Exploration)

Satisfied → **Okay**

Not satisfied → **Counterexample**

PROMELA (1/5)

- Process Meta Language
- Describes the behavior of systems of potentially interacting processes.
- Processes , Objects, Message channels

PROMELA (2/5)

- Process
  - Is instantiations of '*proctype*'
  - defines behavior of system.
  - is consisted of declaration+statements
- Data Object
  - Basic data types : bool, byte, mtype, pid, int, etc.
  - Two levels of scope only : Global and Process local
  - Data structure using '*typedef*'
  - Array

PROMELA (3/5)

- Message channels
  - model the exchange of data between processes
  - declared by '*chan*' .

PROMELA (4/5)

- Statements
  - Assignments and Expressions
  - Deterministic steps
  - Non-deterministic steps
  - Selection
  - Repetition
  - I/O
- Communication
  - Message channel

PROMELA (5/5)

```
chan STDIN ;
proctype Euclid(int x, y){
    do
    :: (x > y) -> x = x – y
    :: (x < y) -> y = y – x
    :: (x == y) -> goto done
    do ;
done:
    printf("answer: %d\n", x) }
```

Message channel & Standard Input
Creating process
Repetition
Selection
Standard output

```
init {
    int a , b ;
    STDIN?a ; STDIN?b;
    run Euclid(a,b)}
```

Declaration of data objects
Communication through channel
Instantiate a proctype

Linear Temporal Logic (1/3)

- Fixed set of atomic formulas and temporal connectives.
- Syntax

$\Phi ::= T \mid F \mid p \mid !\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid X\Phi \mid \Phi U \Phi$

Linear Temporal Logic (2/3)

an infinite word $\xi = x_0 x_1 \ldots$ over alphabet P(Prop)

$\xi \models q$ iff $q \in xo$, for $q \in P$,

$\xi \models !\Phi$ iff not $\xi \models \Phi$,

$\xi \models \Phi_1 \wedge \Phi_2$ iff $\xi \models \Phi_1$ and $\xi \models \Phi_2$

$\xi \models \Phi_1 \vee \Phi_2$ iff $\xi \models \Phi_1$ or $\xi \models \Phi_2$

$\xi \models X\Phi$ iff $\xi_1 \models \Phi$

$\xi \models \Phi_1 U \Phi_2$

iff there is an $i \geq o$ s.t. $\xi_i \models \Phi_2$ and $\xi_j \models \Phi_1$ for all $o \leq j < i$

Linear Temporal Logic (3/3)

- Safety Property
- Liveness Property

Finite State Program

- **$P = (Q, q_o, R, V)$**
  - Q : a finite set of states.
  - q_o : initial state
  - $R \subseteq Q \times Q$: accessibility relation,

    allowing non-determinism.

    Assume that R is total so that a terminate d execution as repeating forever its last st ate.
  - $V : Q \rightarrow P(\text{Prop})$

Büchi automaton

A generalized Büchi automaton $\mathbf{A = (\Sigma, Q, Q_o, \rho, F)}$ where

- Σ : **alphabet**
- Q : a finite set of states,
- $Q_o \subseteq Q$: initial states
- $\rho \subseteq Q \times \Sigma \times Q$: transition relation
- $F \subseteq P(P(Q))$: a set of sets of accepting states.

An accepting execution σ is an execution such that for each acceptance set $\mathbf{F_i \in F}$, there exists at least one state $q \in F_i$ that appears infinitely often in σ.

Automata-Theoretic Software Verification

A finite state program $\mathbf{P=(W,w_o,R,V)}$ can be viewed as a Büchi automaton

$\mathbf{A_p=(\Sigma,W,\{w_o\},\rho,W)}$ where $\Sigma=P(\text{Prop})$

$s' \in \rho(s,a)$ iff $s \rightarrow s'$ and $a=V(s)$

➔Any infinite run of the automaton is accepting

Automata-Theoretic Software Verification

- Global reachability graph : Asynchronous product of processes
- Requirement properties

Automata-Theoretic Software Verification

- For a finite-state program P and LTL formula Ψ ,
- There exists a Büchi automaton A_ψ that accept exactly the computations satisfying ψ.
- The verification problem is to verify that all infinite words accepted by the automaton A_P satisfy the formula Ψ.
- $L_\omega(A_P) \subseteq L_\omega(A_\psi)$ ~ $L_\omega(A_P) \cap L_\omega(\neg A_\psi) = \{\}$

Automata-Theoretic Software Verification

$L_\omega(A) = L_\omega(A_1) \cap L_\omega(A_2)$

let $A_1=(\Sigma,Q_1,Q^0_1,\rho_1,F_1)$, $A2=(\Sigma,Q_2,Q^0_2,\rho_2,F_2)$

Let $A=(\Sigma,Q_1,Q^0,\rho,F)$ where

$Q=Q_1 X Q_2 X\{1,2\}$, $Q0=Q^0_1 X Q^0_2 X\{1\}$, $F=F_1 X Q_2 X\{1\}$

$(q', t', j) \in \rho((q,t,i),a)$ if $s' \in \rho_1(q,a)$, $t' \in \rho_2(q,a)$ and $i=j$

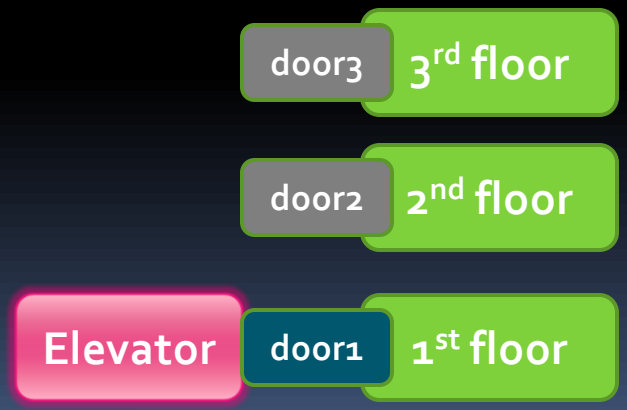
unless $i=1$ and $q \in F_1$, in which case $j=2$

or $i=2$ and $t \in F_2$, in which case $j=1$.

The acceptance condition guarantee that both tracks visit accepting state infinitely often if and only if it goes infinitely often through $F_1 X Q_2 X\{1\}$.

Simple Elevator

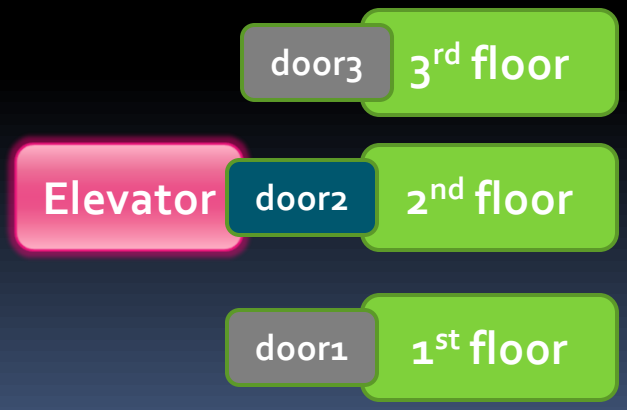
- 3 floor, 1 elevator
- The elevator goes up until 3rd floor and then goes down until 1st floor.
- Each floor has its door to elevator. Each door may open when elevator is at the same floor.



from "System and Software Verification" by B'erard et al

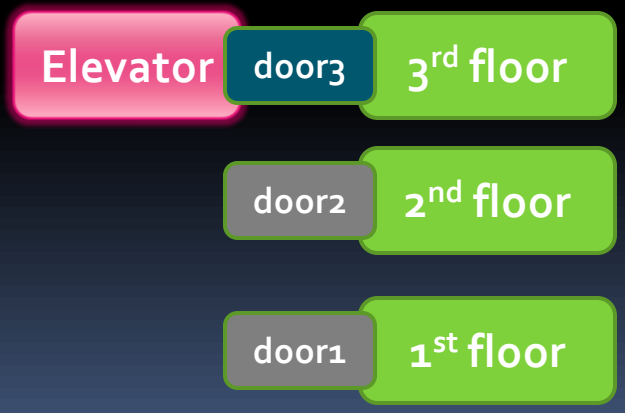
Simple Elevator

- 3 floor, 1 elevator
- The elevator goes up until 3rd floor and then goes down until 1st floor.
- Each floor has its door to elevator. Each door may open when elevator is at the same floor.



Simple Elevator

- 3 floor, 1 elevator
- The elevator goes up until 3rd floor and then goes down until 1st floor.
- Each floor has its door to elevator. Each door may open when elevator is at the same floor.



Simple Elevator::C

```
#define OPENED 1
#define CLOSED 1
sem_t openclosedoor[3] ;
static byte whatfloor ;
static byte doorisopen[3] ;

void door(byte floor)
{
   while(1) {
      sem_acquire(openclosedoor[floor-1]) ;
        doorisopen[floor-1] = OPENED ;
        doorisopen[floor-1] = CLOSED ;
        sem_release(openclosedoor[floor-1]) ;
     }
}
```

```
void elevator()
{
   byte floor = 1 ;

   while(1) {
      if ((rand() % 2) == 0) {
            if (floor != 3) floor++ ;
            else if (floor != 1) floor-- ;
      }
      else {
         sem_release(openclosedoor[floor-1]) ;
         sem_acquire(openclosedoor[floor-1]) ;
      }
   }
}
```

Simple Elevator::C

```
byte args[3] ;
void main()
{
    int i ; byte * temp ;        pid_t pid ;
    sem_init(openclosedoor[0], 0) ;
    sem_init(openclosedoor[1], 0) ;
    sem_init(openclosedoor[2], 0) ;

    for (i = 0 ; i < 3 ; i++) {
        args[i] = I ;
        pid = thread_create(door, &(args[i])) ;
        thread_join(pid) ;
    }
}
```

Simple Elevator::PROMELA

```
bit doorisopen[3] ;
chan openclosedoor=[0] of {byte, bit}

proctype door(byte i)
{
    do
    :: openclosedoor?eval(i), 1 ;
      doorisopen[i-1] = 1 ;
      doorisopen[i-1] = 0 ;
      openclosedoor!i,0
    od
}
```

```
proctype elevator()
{
        byte floor = 1 ;
        do
        :: (floor != 3) -> floor++
        :: (floor != 1) -> floor—
        :: openclosedoor!floor,1 ;
          openclosedoor?eval(floor),0;
        do
}

init {
  atomic{
    run door(1) ; run door(2) ;
    run door(3) ; run elevator() ;
  }
}
```

Simple Elevator::Verification

- assert(

doorisopen[i-1]&&!doorisopen[i%3]&&!doorisopen[(i+1)%3]);

```
#define open1 doorisopen[0]      #define open2 doorisopen[1]
#define open3 doorisopen[2]      #define close1 !doorisopen[0]
#define close2 !doorisopen[1]    #define close3 !doorisopen[2]
```

- [](open1 -> X closed1)
- [](open2 -> X closed2)
- [](open3 -> X closed3)
- <>(open1 || open2 || open3)

Further Reading

- ω-language
- Partial order reduction
- Memory management technique in SPIN

References

[1] The Model Checker SPIN, G.J.Holtzmann.

[2] The SPIN model checker, G.J.Holtzmann.

[3] Systems and Software Verification, Berard et al.

[4] Simple On-the-fly automatic verification of Linear temporal logic, Vardi et al.

[5] An Automata-Theoretic Approach to Linear Temporal Logic, M.Y.Vardi.

[6] Moonzoo Kim's lecture notes on CS750 Fall2006, CS KAIST.

Discussion