

Logic Summer School, ANU Dec 2003  
Formal Methods 1: Algorithmic Verification  
Ron van der Meyden (UNSW/NICTA)

## Algorithmic Verification

Verification *by* algorithms

Applications:

1. hardware design
2. communications protocols
3. embedded systems
4. cryptographic protocols

## Overview of Course

1. Concurrent programs, explicit state verification
2. Linear time temporal logic, Buchi automata
3. Partial Order Optimization, Bounded Model Checking
4. Branching time temporal logic, BDD optimization
5. Model Checking Epistemic Logic

We will only scratch the surface of this topic. Some key references for this area are:

1. E. Clarke, O. Grumberg and D. Peled, *Model Checking*, MIT Press 1999
2. *The Spin Model Checker*, G. Holzmann, 2003
3. R.P. Kurshan *Computer Aided Verification of Coordinating Processes* (The automata theoretic approach) Princeton Series in CS, 1994
4. *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Semantics, MIT Press 1994
  - (a) E.A. Emerson, *Temporal and Modal Logic*,
  - (b) W. Thomas, *Automata on Infinite Objects*

Texts on Proof Theoretic verification of concurrent systems:

1. Z. Manna and A. Pnueli, The temporal Logic of Reactive and Concurrent Systems, Specification, Springer Verlag 1992
2. Z. Manna and A. Pnueli, Temporal Verification for Reactive Systems, Safety, Springer Verlag 1995
3. Concurrency Verification: Introduction to Compositional and Noncompositional Methods Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers Cambridge University Press, 2001.
4. Verification of Sequential and Concurrent Programs, K.R Apt, E.R Oldero,

**Lecture 1: Concurrent Programs, Explicit state model checking**

## Mutual Exclusion Protocols

Two processes both want to access a shared resource - if they do so at the same time, something bad happens:

Example:

resource = a printer,  
bad thing = our documents get shuffled!

Each process has a *critical section*, during which it requires exclusive access to the resource.

A mutual exclusion protocol ensures *the processes are not simultaneously in their critical sections*

**do** Non critical section ; critical section **od**

Requirements for mutual exclusion of two processes (more later):

1. (Mutual Exclusion) The two processes are not in their critical sections simultaneously.

A naive approach: a process raises a flag when it wishes to go into its critical section, and waits if the other also wishes to go in. i.e. for process 0:

```
do
  Non-critical section;
  flag0 := true;
  while flag1 = true do skip;
Critical section;  flag0 := false
od
```

Problem: deadlock. If both are trying to go critical, both wait forever.

### Peterson's Algorithm

(A new solution to Lamport's concurrent programming problem, ACM Trans on Prog Lang and Systems, 5(1) 1983 pp. 56-65)

Idea: In addition to raising a flag, the processes write their name to a shared variable *last*. If both are trying to go critical, the first to write to this variable is the first to go critical. If both are trying, a process can tell if it was first by checking that the *last* variable is set to the other process.

### Peterson's Mutual Exclusion Protocol

```
bool flag[2];
bool last;

proctype process0()
{
  do ::
    nc1:  flag[0] = 1;
    nc2:  last = 0;
    try:  (flag[1] == 0 || last == 1);
    crit1: skip; /* critical section */
    crit2: flag[0] = 0
  od
}
```

```
proctype process1()
{
  do ::
    nc1:  flag[1] = 1;
    nc2:  last = 1;
    try:  (flag[1] == 0 || last == 0);
    crit1: skip; /* critical section */
    crit2: flag[1] = 0
  od
}

init {run process0(); run process1()}
```

## Explicit State Model Checking of Safety Properties

Input: A set of processes operating in parallel

Question: is it possible to reach a state where some property  $P$  holds ( $P$  a predicate over the shared and local variables)

Approach: check every state!

This may work provided the problem is constrained enough that there is a finite set of states

Even finite state systems are often hard to reason about!

## Transition Systems

A *transition system* is a tuple  $M = \langle S, S_0, \rightarrow, L \rangle$  where

1.  $S$  is a set, of *states*
2.  $S_0$  is a subset of  $S$ , the *initial states*
3.  $\rightarrow \subseteq S \times S$  is a *transition relation*
4.  $L : S \rightarrow \mathcal{P}(\text{Prop})$  is an *interpretation*

We write  $s \models p$  if  $p \in L(s)$

## Constructing a transition system from a Parallel Program

Let  $V = \{v_1, \dots, v_n\}$  be a set of shared variables

Let  $L_i$  be the set of program locations of process  $i = 1 \dots k$

A *state* is an n-tuple of the form

$$[v_1 = val_1, \dots, v_n = val_n, loc_1 = l_1, \dots, loc_k = l_k]$$

where each  $val_i$  is a value of the appropriate type for  $v_i$  and each  $l_i \in L_i$  is a program location.

## Example

For Peterson's protocol,

$$[flag[0] = 0, flag[1] = 1, last = 0, loc_1 = crit1, loc_2 = crit2]$$

is a state.

## Transition Relation

Define the transition relation  $\rightarrow$  on states by  $s \rightarrow t$  if there exists a process  $i$  such that  $s \models loc_i = l_i$ , and program  $i$  is able to execute a single step in state  $s$ , and the effect is to produce state  $t$ .

This is an *interleaving semantics*

## Model Checking Safety properties

Safety = “nothing bad happens”

Let  $\phi$  be a predicate over states e.g,

$\neg loc_0 = crit \wedge loc_1 = crit$

A concurrent program  $Prog = (S_0, P_1 || \dots || P_n)$  satisfies  $\phi$  if for all reachable states  $s$  of the transition system  $M_{Prog}$ , we have  $s \models \phi$ .

## An algorithm for Model checking safety properties

Depth First Search! Mark states you have already visited

For each  $s$  in  $S_0$  do DFS(s,phi)

DFS(s,phi) =

if  $s$  is marked then return true;

mark  $s$ ;

if  $s \models \phi$  then return false

else for each  $t$  such that  $s \rightarrow t$ ,

do DFS( $t$ ,phi)

if any DFS( $t$ ,phi) returns false, return false,

else return true

A state  $s$  of  $M = \langle S, S_0, \rightarrow, L \rangle$  is *reachable* if there exists a sequence

$s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n = s$

where  $s_0 \in S_0$ .

Another Mutual exclusion protocol:

**Process 1: Do nothing**

**Process 2: Do nothing**

This satisfies the safety property we stated:

(Mutual Exclusion) The two processes are not in their critical sections simultaneously.

But it is not a very interesting/useful solution.

We want not just “nothing bad happens” but also “something good happens”

## Background

Prior, 1950-60's: studied *tense logic*, i.e. normal modal logic with operators such as

1.  $G\phi$ , meaning  $\phi$  will hold at all future times
2.  $F\phi$ , meaning  $\phi$  will hold at some future time
3.  $P\phi$ , meaning  $\phi$  will hold at some past time

Semantics: Kripke frames, with accessibility relation  $R$  such that  $uRv$  represents “ $v$  is an instant of time in the future of instant  $u$ ”

Motivation: Philosophical-Linguistic Analysis

Pnueli, The temporal Logic of Programs, Proc 18th IEEE Symp on Foundations of Computer Science, 1977 proposed to apply temporal logic to reasoning about concurrent/reactive programs

The proposal led to a rich area of research encompassing

1. new operators, logics, axiomatizations
2. expressiveness studies
3. algorithmic properties of temporal logics
4. verification methodology/proof rules for concurrent/reactive systems
5. automated and semi-automated verification tools

Pnueli awarded Turing Award 1996 “For seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification.”

More Requirements for mutual exclusion of two processes:

1. (Liveness/Deadlock Freedom) At all times, one of the processes must be able to proceed
2. (Fairness/Lack of starvation) If a process is trying to go into its critical section, it is not forever prevented from doing so.

For these, we need a way of talking about what happens over time ...