

Automatic Discovery of Mutual Exclusion Algorithms

PODC 2003

- Yoah Bar-David
- Gadi Taubenfeld

Ercan UÇAN

CS 523

April 5, 2006

Outline

- Introduction
 - What is this paper about?
 - Overview of the system generated
- System Components
 - Algorithm Generator
 - Algorithm Verifier
 - Optimizations
- Tests and Results
- Discussion

Intro:

The Mutual Exclusion Problem Recap

- To design an algorithm that guarantees mutually exclusive access to a critical section among racing processes.
- Requirements
 - Mutual Exclusion
 - Deadlock-freedom
 - Starvation-freedom (stronger requirement)



Intro:

Automatic Discovery

- Methodology for automatic discovery of mutex algorithm is proposed and developed.
- For a problem P
 - 1) Write a model checker M for P .
 - 2) For a given(restricted) programming language, write a program to generate all syntactically possible algorithms.
 - 3) Check each algorithm using M whether it solves P

Intro:

System architecture overview

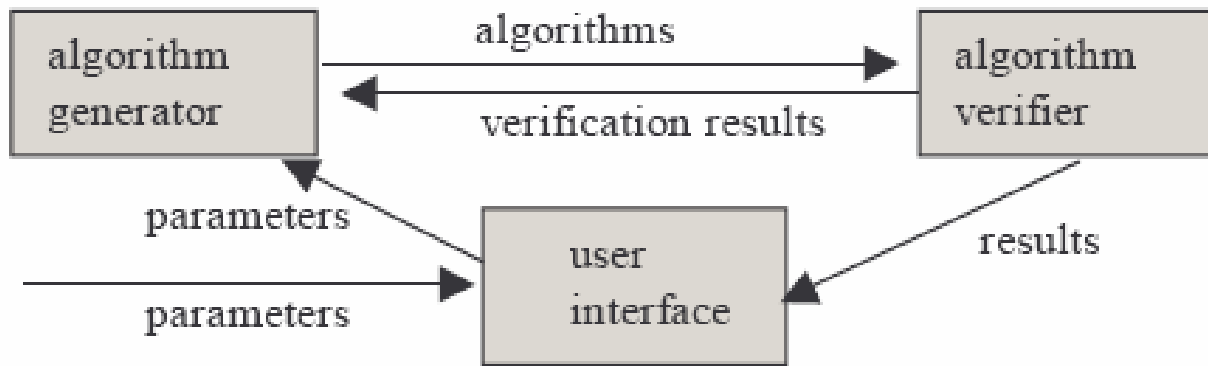


Fig. 1. System architecture for a single computer

- User Interface
- Algorithm Generator
- Algorithm Verifier

System Components: Algorithm Generator

- First attempt was to generate algorithms using assembler-like language
- Simple high-level language was designed with parameters:
 - # of processes and shared variables
 - # of entry and exit commands
 - Max value for each shared variable
 - Are complex conditions allowed, if so which (and, or, xor)
 - Are multi-writer variables allowed

System Components: Algorithm Generator

- Steps of algorithm generation
 - Set the parameters
 - Enumerate all possible variables, assignments, conditions and control structures(command type to a line of code).
 - First possible algorithm generated
 - After verification, # of lines incremented, next possible command found...

System Components: Algorithm Generator

- Language used to generate algorithms: Language elements

Constants	Integers, from zero to the highest allowed value for a variable.
Relative constants	Contain a process number: me (my process number, zero for first process), next (successor process number), prev (predecessor's number).
Simple variables	Integers, can have a value from zero to the highest allowed value for a variable.
Arrays	One-dimensional array of simple variables. The array size is the same as the number of processes.
Referencable variable	Either a simple variable, or an array variable with an index. The index can be a constant, a relative constant or a simple variable
Simple conditions	A comparison between 2 variables or a variable and a constant. Comparison operators are = (equals) and != (not equal)
Complex conditions	2 simple conditions, related with and , or or xor

System Components: Algorithm Generator

- Language used to generate algorithms: Statements

Assignment statement	<i>referencable-variable = constant</i> (e.g. $v=0$ or $a[1] = 0$) <i>referencable-variable = referencable-variable</i> (e.g. $v=q$ or $a[1]=b[v]$)			
if and while statements	if <i>condition</i> <i>statement(s)</i> endif	if <i>condition</i> <i>statement(s)</i> else <i>statement(s)</i> endif	while <i>condition</i> <i>statement(s)</i> endwhile	while <i>condition</i> <i>statement(s)</i> endwhile

System Components: Algorithm Generator

- Sample code generated by the algorithm generator:

Peterson's algorithm for 2 processes

1 a[me] = 1

2 turn = next

3 while a[next] = 1 and turn = next

4 endwhile

5 critical section

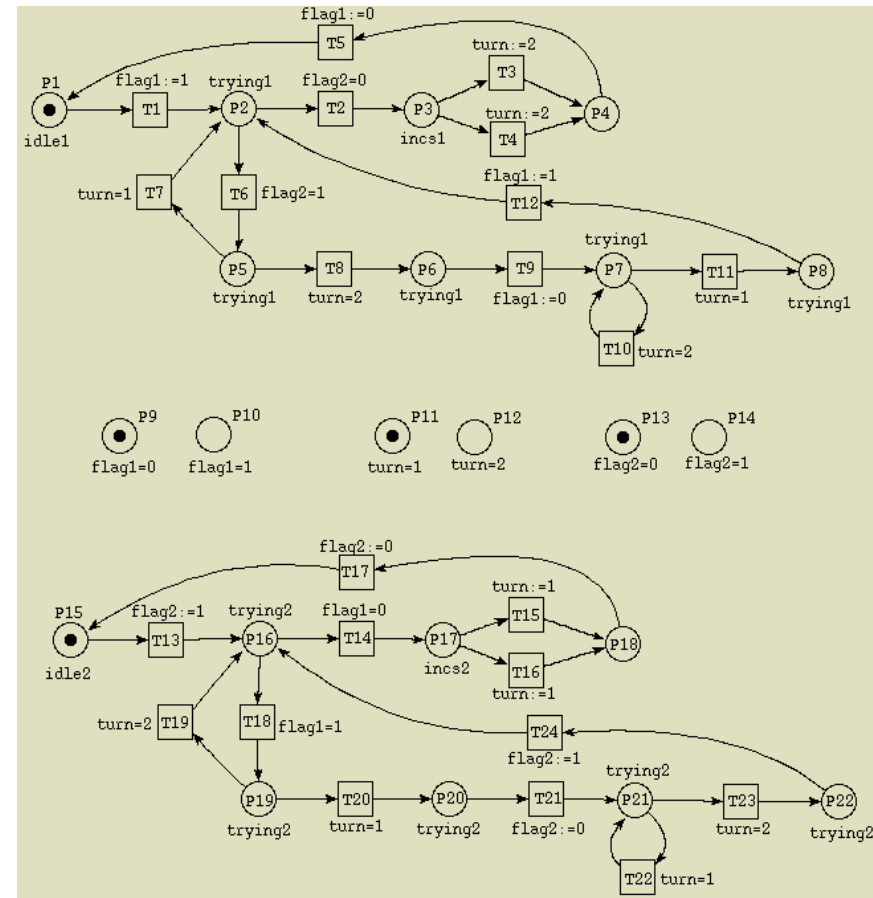
6 a[me] = 0

System Components: Algorithm Verifier

- Model checking on finite state concurrent systems.
- Very fast verifier for mutual exclusion is developed.
- Verification language was designed.
 - Low-level assembler like language.
 - Instructions represented by a triplet: 1 op code, 2 optional operands
 - Each process have 3 local variables: program counter, general purpose register, index register

System Components: Algorithm Verifier

- State Transition Graph
 - All reachable states are added to the graph from the initial condition
 - Validation checks are run on the model.



System Components: Algorithm Verifier

- Test of Mutual Exclusion
 - Reject if more than one processes is in the critical section(CS)
- Test of Deadlock-freedom
 - Reject if there is a cycle where all active processes executed at least one instruction and no process is in its CS.
- Test of Starvation-freedom
 - Not satisfied if there is a cycle where all active processes executed at least one instruction and some process was not in the CS during any state of the cycle.

Optimizations

- Optimizations during algorithm generation
 - Do not generate syntactically incorrect algorithms
 - Every open block is properly closed
 - Do not generate equivalent conditions
 - $a = 1$ is equivalent to $1 = a$
 - Do not generate constant conditions
 - $(a = 0 \text{ and } a = 1)$
 - Do not allow consecutive assignments to the same variable
 - $a = 1$
 $a = 0$
 - Do not generate relative constant 'prev' for 2 processes

Optimizations

- Optimizations during verification
 - Incremental state graph construction
 - Largest valid sub-graph
 - Stop building state graph on first error
 - On-the-fly mutual exclusion check
 - Minimize number of states
 - New state generated if last executed command accessed a global variable

Optimizations

- Interactive Optimizations

- Skip generation of alternatives for complex conditions that were not executed
 - if(a = 1 or b = 2)
- Skip generation of alternatives for statements that are not executed
 - 1 a = 0
 - 2 if a = 1
 - 3 b = a
 - 4 endif
 - 5 critical section
 - 6 a = 0

Optimizations

- Optimizations specific to mutual exclusion
 - Check 'solo' runs first
 - Must have 'while' in the entry code
 - Must have assignment statement in the entry code and in the exit code

Tests and Results

- System was run to find algorithms for two processes using 2,3,4,5, or 6 shared bits on a Pentium 4/1.6GHz PC.

User-defined parameters				Results			
Shared bits	Entry commands	Exit commands	Complex conditions	Starvation freedom	Tested algorithms	Correct algorithms	appx. running hours
2	6	1	Yes		7,196,536,269	0	216
2	7	1			846,712,059	66	39
3	4	1	Yes	Yes	25,221,389	105	0.4
3	6	1		Yes	1,838,128,995	10	47
4	4	1	Yes	Yes	129,542,873	480	1
4	5	1			129,190,403	56	1
4	6	1		Yes	*900,000,000	80	12
5	5	1			*22,000,000	106	0.4
6	5	1			*70,000,000	96	1

Tests and Results

- Two shared bits, simple conditions

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	27,372	0
4	2	44,340	0
5	1	925,389	0
5	2	1,235,778	0
6	1	28,522,988	0
7	1	846,712,059	66

```
1 a[me] = 1
2 while a[next] = 1
3   while a[0] = me
4     a[me] = 0
5   endwhile
6   a[me] = 1
7 endwhile
8 critical section
9 a[me] = 0
```

- Shortest solutions found for this case has 7 entry and 1 exit commands.
- None of the 66 satisfy deadlock-freedom

Tests and Results

- Three shared bits, simple conditions

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	287,579	0
4	2	493,073	0
5	1	24,124,934	0
5	2	36,636,722	0
6	1	1,838,128,995	8 starvation-free 2 deadlock-free

3 bits, simple conditions starvation-free	3 bits, simple conditions deadlock-free
1 a[me] = next	1 a[me] = 1
2 if b = next	2 while b != a[next]
3 b = me	3 while a[b] = me
4 while b = a[next]	4 b = a[0]
5 endwhile	5 endwhile
6 endif	6 endwhile
7 critical section	7 critical section
8 a[me] = me	8 a[me] = 0

- Shortest solutions found for this case has 6 entry and 1 exit commands.
- Shorter than Dekker's starvation free algorithm. Dekker's has 9 entry and 2 exit commands when ported.

Tests and Results

- Four shared bits, simple conditions

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct algorithms
4	1	915,350	0
4	2	1,270,897	0
5	1	129,190,403	56 deadlock-free
6	1	*900,000,000	80 starvation-free

4 bits, simple conditions, deadlock-free	4 bits, simple conditions, starvation-free
1 a[me] = 1	1 a[me] = 1
2 b[0] = 0	2 b[me] = me
3 while b[me] != a[next]	3 a[1] = 0
4 b[next] = me	4 a[me] = a[0]
5 endwhile	5 while b[next] != a[1]
6 critical section	6 endwhile
7 a[me] = 0	7 critical section
	8 b[me] = next

- Shortest deadlock-free solutions found for this case has 5 entry and 1 exit commands.
- Shortest starvation-free solutions found for this case has 6 entry and 1 exit commands.

Tests and Results

- Three shared bits, complex conditions

Parameters		Results	
Entry comm.	Exit comm.	Tested algorithms	Correct alg.
3	1	75,496	0
3	2	492,707	0
4	1	25,221,389	105 (all s.f.)

Peterson's algorithm for 2 processes

```
1 a[me] = 1
2 turn = next
3 while a[next] = 1 and turn = next
4 endwhile
5 critical section
6 a[me] = 0
```

```
1 a[me] = 1
2 b = me
3 while a[next] = me
  xor b = 0
4 endwhile
5 critical section
6 a[me] = 0
```

- Peterson's algorithm is the shortest algorithm found.
- All the 105 algorithms satisfy starvation-freedom and all remaining 104 are variants of Peterson's algorithm.

Discussion

- Open issues
 - More optimizations and heuristics
 - Application to other synchronization problems
 - Why some sets of parameters have no starvation-free algorithms?
 - Are there meaningful classes that solutions can be reduced to?
- Evaluation
 - Showed that approach demonstrated is feasible.
 - No test cases for more than 2 processes.

Questions?

- Thanks for your attention!