



Concurrency and Correctness in SPIN

Literature: Holtzmann ch 8 and 11

Mads Dam

This Lecture

Go through the technical foundations of the SPIN tool:

- The internal SPIN model for transition systems
- Asynchronous product
- Algorithms for state space traversal
- Invariants
- Progress and acceptance cycles
- Never claims
(= Buchi automata + valid end states)

Transition System Specifications

Similar to Peled

Data objects: Determines domain and initial value
– Default is 0

State vector: Vector $(x_1=v_1, \dots, x_n=v_n)$ of data object values

Transition specification/atomic action:
 $\phi \rightarrow (x_1, \dots, x_n) := (e_1, \dots, e_n)$
 ϕ and each e_i is side effect free expression

May contain separate control component (fsm)

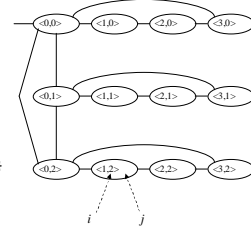
Generated (expanded) state space as in Peled

Asynchronous Product - Example

```
byte i ;
byte j ;

active proctype p1()
{
  do
  :: i = (i + 1) mod 4
  od
}

active proctype p2()
{
  do
  :: atomic {
  i=0 -> j=(j+1) mod 3
  }
  od
}
```



Basic Verification Algorithm

Depth first search (DFS)

Fact: $search(q_0)$ always terminates and when it does, it will have visited all states reachable from q_0

Problem: Works on generated state space, so this must have been previously computed

Sol'n: Redesign algorithm to take an asynchronous product, and compute states on the fly

```
start()
{
  addstatespace(q0)
  pushstack(q0)
  search()
}

search()
{
  q = topstack()
  whenever q -> q'
  if inspacespace(q') == false
  {
    addstatespace(q')
    pushstack(q')
    search()
  }
  popstack()
}
```

On-the-Fly Depth First Search

```
Start()
{
  addstatespace(q0)
  pushstack(q0)
  search()
}

search()
{
  q = topstack()
  for all i in {1, ..., n}
  {
    si := Control(q,i)
    v := Data(q)
    whenever si -> si' (sic!)
    if Cond(alpha,q) == true
    {
      q' := Update(q,i,si',alpha)
      if inspacespace(q') == false
      {
        addstatespace(q')
        pushstack(q')
        search()
      }
    }
  }
  popstack()
}
```

Operates on asynchronous product T_1, \dots, T_n
Q has control state (s_1, \dots, s_n)
v has shape $(x_1=v_1, \dots, x_n=v_n)$
 $Control(q,i) = s_i$ (i'th control state)
 $Data(q)$: Value of state vector
 $Cond(\alpha,q)$: Firing condition of α , evaluated at q
 $Update(q,i,s,\alpha)$: Changes i'th control state in q to s, and applies multi-assignment in α to data objects

Correctness Claims

State properties: Conditions to hold in specific states – maybe all

- State assertions
Whenever local state s_i is reached, $k \neq 0$
- System invariants
In all reachable states, $i < 4$ or $j > 3$

Path properties: Conditions to hold of execution sequences = run = computation paths

- Progress properties
- Fairness properties
- "a request packet is sent infinitely often"

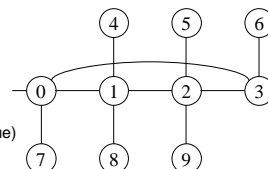
State Properties -Example

Program:

```
do
  :: i = (i + 1) mod 4
od
```

with $V = \{0,1,2,\dots,9\}$, $v_0 = 0$

State property:
In all reachable states $i < 4$ (true)



Proving Invariants

Task: Specify that state property p holds in all reachable states

Add edge label

$\langle 1, \text{assert}(p) \rangle$

$\text{assert}(p)$ evaluates p at the given state

DFS can signal if assertion is violated

Add edge (s,s) labelled by $\langle 1, \text{assert}(p) \rangle$ to all states s

Possible implementation: Add

active proctype $\text{inv}()$

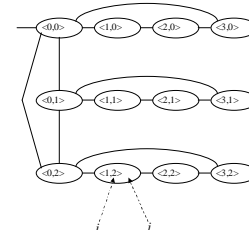
```
{
  do
    :: assert{p}
  od
}
```

Asynchronous product will assert p in all reachable states

Path Properties - Examples

For all paths:

- $\llbracket \langle \rangle i = 0 \text{ (true)} \rrbracket$
- $\llbracket \langle \rangle (i = 1 \vee j = 1) \text{ (true)} \rrbracket$
- $\llbracket \langle \rangle j = 0 \text{ (false)} \rrbracket$
- $(\llbracket \langle \rangle j = 0 \rrbracket \wedge (\llbracket \langle \rangle j = 1 \rrbracket \rightarrow \llbracket \langle \rangle j = 2 \rrbracket))$



Marked Transition Systems

Two options for checking path properties in SPIN:

1. Annotate states explicitly to talk about valid endstates, progress cycles and acceptance cycles.
2. Represent temporal constraints as Buchi automata and check them through a synchronous product construction

Marked LTS: Transition system augmented with labelling of control states as:

- Valid end states
States that should count as normally terminated
- Progress states
States that should be traversed i.o. (e.g. successful reception)
- Acceptance states
States that should *not* be traversed i.o. (e.g. retransmission)

Marking Products

Control state for product marked by marking local states:

- $(s_1, \dots, s_n) \in \text{End}$ iff for all $i: 1 \leq i \leq n$, $s_i \in \text{End}_i$
- $(s_1, \dots, s_n) \in \text{Prg}$ iff $s_i \in \text{Prg}_i$ for some $i: 1 \leq i \leq n$
- $(s_1, \dots, s_n) \in \text{Acc}$ iff $s_i \in \text{Acc}_i$ for some $i: 1 \leq i \leq n$

Note: Fairness *not* captured by this definition:

- Can have progress cycle such that only one process progresses
- But requiring $\forall i: 1 \leq i \leq n$, $s_i \in \text{Prg}_i$ is too strong – such states might never be visited

No "nesting" of progress/acceptance labelling possible

- Such as: Discount acceptance cycles by progress cycles

Important to design model to reflect above

Verifying Path Properties

Deadlock detection: Is there a state in asynchronous product from which no transitions are possible, but the state is not a valid endstate?

Progress cycle: Is there an infinite trace that visits a progress state a finite number of times only?

Or: Is there a reachable non-progress state which is reachable from itself along a path which does not contain progress states?

Acceptance cycle: Is there an infinite trace which visits a state in Acc infinitely often?

Or: Is there a reachable accepting state which is reachable from itself?

On-the-Fly Deadlock Detection

```

search()
{
  boolean terminating = true
  integer inendstate = n
  q = topstack()
  for all i in {1,...,n}
  { s_i := Control(q,i) ;
    v := Data(q) ;
    if s_i # End(inendstate-)
      whenever s_i -> s_i'
      if Cond(alpha,q) == true
      { q' := Update(q,i,s_i',alpha) ;
        if instatespace(q) == false
        { addstatespace(q)
          pushstack(q)
          search()
          terminating = false ; }
        if terminating == true and inendstate #
          n
          { report deadlock state; printstack()
            popstack() }
      }
  }
}
    
```

n: Number of processes

terminating: Initially true. Set to false when successor state is found

inendstate: Decrement value by one for each component which is not in end state

Deadlock found when:

- *terminating* == true
- *inendstate* # n

All Deadlocks Are Found

Theorem. When the algorithm terminates it has visited all states reachable from the initial state

Proof: (By contradiction)

- Assume there is a reachable state which isn't visited
- Since the state is reachable there is some path which contains it
- In this path, let q be the last state visited and q' the first state not visited
- We find some i such that

$$s_i \rightarrow s_i'$$

$$s_i := \text{Control}(q,i)$$

$$v := \text{Data}(q)$$

$$\text{Cond}(\alpha,q) == \text{true}$$

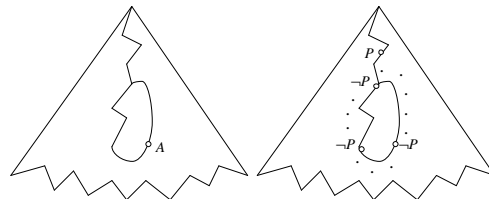
$$q' := \text{Update}(q,i,s_i',\alpha)$$

- Since the algorithm did not visit q', *in_statespace*(q') == true
- But this can only be the case when the algorithm has already visited q'

Cycle Detection

(Büchi) Acceptance Cycle

Non-progress Cycle



Acceptance Cycle Detection

Idea:

- Q: Is there a cycle originating in reachable accepting state back to itself?
- If accepting state q is found, initiate second traversal from that state
- Use two state spaces
- If second traversal reaches q shout GOT IT
- The cycle is on the stack
- Variation for non-progress cycles: Start second search from non-progress states, exit when progress state encountered

Progress cycle detection is similar, see Holtzmann's book

On-the-Fly Acceptance Cycle Detection

```

search()
{
  integer accepting = false
  q = topstack()
  if q == seed
  { acceptance cycle!; printstack(); exit()
  }
  for all i in {1,...,n}
  { s_i := Control(q,i) ;
    v := Data(q) ;
    if s_i is accepting(accepting := true)
      whenever s_i -> s_i'
      if Cond(alpha,q) == true
      { q' := Update(q,i,s_i',alpha) ;
        if instatespace(q) == false
        { addstatespace(q)
          pushstack(q)
          search() }
        if seed == nil and accepting == true
        { seed = q ; secondstatespace();
          search();
          firststatespace(); seed = nil }
        popstack() }
  }
}
    
```

firststatespace,
secondstatespace: Switches between state spaces

seed: Holds accepting state during second traversal

Cost: 2 x time, 2 x space

Theorem: An Acceptance Cycle Is Found If One Exists

Proof: (By contradiction)

- Let q_a be the (postorder) first acceptance state reachable from itself which is encountered in the DFS
- Second search from q_a can only fail to close the loop if another q_n exists which is already visited
- q_a is reachable from q_n and vice versa
- q_a is reachable also from some acceptance state q_n from where second traversal started
- q_n can not be on the stack since the postorder traversal of state space 1 would then cause the second search from q_n to take place after the second search of q_a
- But the path from q_n to q_a must intersect the stack, and so q_n must be reachable from q_a
- But then there is a path from q_n to q_n , and q_n will be visited before q_a in a postorder traversal
- This is a contradiction

2005 Mads Dam IMIT, KTH

19

2G1516 Formal Methods

LTL in SPIN

Avoid complementation of Buchi automata

Specify instead "never claim" ϕ , temporal property which should *not* hold of model

Convert ϕ to Buchi automaton

Introduce synchronous product \times

Product implements restricted intersection operation

- One argument to be unmarked
- Essentially: All states accepting, no end states, no progress states

Perform acceptance cycle test on product automaton

2005 Mads Dam IMIT, KTH

20

2G1516 Formal Methods

Never Automata, II

Never automata: Marked transition system with

- Empty set of progress states
- Nonempty set of acceptance states
- No side-effects

Path acceptance: Path ξ is accepting if either

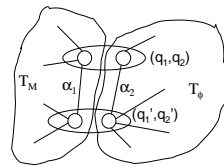
- ξ is finite and the last state of ξ is a valid end state
- ξ is infinite and visits an accepting state infinitely often

2005 Mads Dam IMIT, KTH

21

2G1516 Formal Methods

Synchronous Product



Product transition label:

$$\phi_1 \wedge \phi_2 \rightarrow (x_1, \dots, x_n) := (e_1, \dots, e_n)$$

Product state (q_1, q_2) :

- End state, if q_2 is end state (in T_ϕ)
- Accepting state, if q_2 is accepting (in T_ϕ)

$$\alpha_1 = \phi_1 \rightarrow x_1, \dots, x_n := e_1, \dots, e_n$$

$$\alpha_2 = \phi_2 \rightarrow () := ()$$

2005 Mads Dam IMIT, KTH

22

2G1516 Formal Methods

Concluding Remarks

With tools like SPIN models that are not too large *can* be exhaustively validated

With state space hashing two orders of magnitude can be gained – but result is approximate

Even so, memory quickly runs out

Overcoming this:

- Use smaller models = more abstraction
- Avoid traversing the global state space – e.g. use invariants
- Analyse systems componentwise – current research topic
- Symbolic treatment of data – also a research topic

2005 Mads Dam IMIT, KTH

23

2G1516 Formal Methods