NuSMV Seminar 2: Advanced Topics

INIT & TRANS, BMC, Paxos, Code Generation

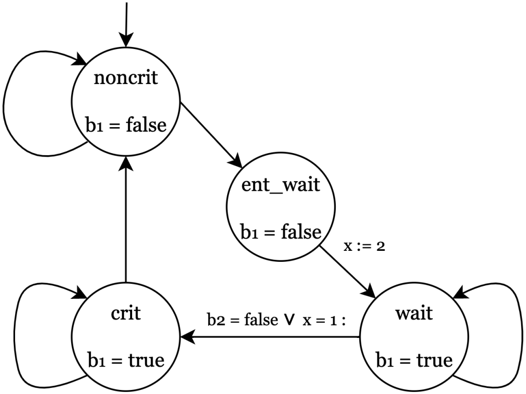
Di Marco, Okwieka

University of Rome "La Sapienza"

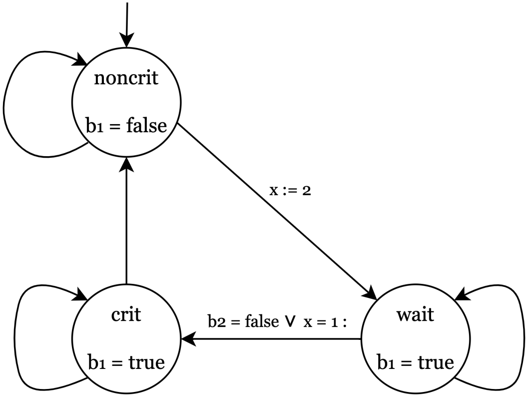
May 8th, 2023

ASSIGN

Peterson's Mutual Exclusion Algorithm



Peterson's Mutual Exclusion Algorithm



NuSMV code

```
MODULE main
VAR
        x   : 1 .. 2;
        pg1 : peterson(1, x, pg2.b);
        pg2 : peterson(2, x, pg1.b);
ASSIGN
        next(x) := case
                        (pg1.state = wait)  : 2;
                        (pg2.state = wait)  : 1;
                        TRUE                : x;
                esac;

FAIRNESS
        pg1.state = crit
FAIRNESS
        pg2.state = crit


LTLSPEC G !(pg1.state = crit & pg2.state = crit)
```

```
MODULE peterson(id, x, other_b)
VAR
        state : { noncrit, wait, crit };
ASSIGN
        init(state) := noncrit;
        next(state) := case
                (state = noncrit)

                     : { noncrit, wait };
                (state = wait) & ((id = x)
                      | !(other_b)) : {
                     wait, crit };
                (state = crit)

                     : { crit, noncrit };
                TRUE

                     : state;
            esac;
DEFINE
        b := (state = wait) | (state = crit);
```

Counterexample

```
-- specification  G !(pg1.state = crit & pg2.state
     = crit) is false
-- as demonstrated by the following execution
     sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    x = 1
    pg1.state = noncrit
    pg2.state = noncrit
    pg1.b = FALSE
    pg2.b = FALSE
  -> State: 1.2 <-
    pg1.state = wait
    pg2.state = wait
    pg1.b = TRUE
    pg2.b = TRUE
  -> State: 1.3 <-
    x = 2
    pg1.state = crit
  -> State: 1.4 <-
    x = 1
    pg2.state = crit
    pg1.b = FALSE
  -- Loop starts here
    ...
```

INIT & TRANS

INIT & TRANS

To be able to perform multiple variable updates in a single step even when nondeterminism is at play we will need to use the INIT and TRANS statements.

Given an arbitrarily complex *propositional formula*, the statements will define the set of **initial** states and **successor** states as all the states that at a certain step, and at a certain assignment of the variables, **satisfy said formulas**.

Peterson fixed

```
MODULE peterson(id, x, other_b)
VAR
        state : { noncrit, wait, crit };
INIT
        state = noncrit;
TRANS
        ( (state = noncrit
                & id = 1) ->
                        (next(state) = noncrit | (
                             next(state) = wait &
                             next(x) = 2)) )

        & ( (state = noncrit
                & id = 2) ->
                        (next(state) = noncrit | (
                             next(state) = wait &
                             next(x) = 1)) )

        & ( ((state = wait & (id=x | !other_b))
                        ) ->
                                (next(state) = wait | next
                                     (state) = crit) )

        & ( (state = crit
                        ) ->
                                (next(state) = crit | next
                                     (state) = noncrit) )

        & ( ( !(state = noncrit)
                & !((state = wait & (id=x | !
                        other_b)))
                & !(state = crit) ) ->
                        (next(state) = state &
                             next(x) = x) );

DEFINE
        b := (state = wait) | (state = crit);
```

Risks

Differently from the case statement, if multiple different assignments for the same variable are implied by our formulas, **NuSMV will not warn us**.

It is also possible to write **contradicting formulas** that will never be *TRUE*, resulting in states with no outgoing transitions.

Best practices

The simplest and *most readable* way to use the TRANS statement is to specify a transition as an **implication**

$$state_i \longrightarrow next(state_i)$$

Best practices

This way we can define transition systems as conjunctions of said implications.

$$
\begin{aligned}
& \mathit{state}_1 \longrightarrow \mathit{next}(\mathit{state}_1) \\
\wedge \quad & \mathit{state}_2 \longrightarrow \mathit{next}(\mathit{state}_2) \\
\wedge \quad & \ldots \\
\wedge \quad & \mathit{state}_n \longrightarrow \mathit{next}(\mathit{state}_n) \\
\wedge \quad & (\neg \mathit{state}_1 \wedge \ldots \wedge \neg \mathit{state}_n) \longrightarrow (\mathit{keep\ current\ assignment})
\end{aligned}
$$

Bounded Model Checking

Bounded Model Checking (BMC)

Biere et al. 1999: *Symbolic model checking without BDDs*

NuSMV uses BDDs for LTL model checking

Bad ordering of variables can lead to storage space explosion

But: no simple way of finding optimal, or even good orderings for all cases

Idea: Use *propositional logic* SAT solver

- No canonical form, no space explosion, thousands of variables
- Nowadays heavily used in other areas $\Longrightarrow$ mature
- Little space use, fast depth-first approach

Bounded Model Checking (BMC)

Biere et al. 1999: *Symbolic model checking without BDDs*

Bounded Model Checking: construct prop. logic formulas ϕ_k with $k \leq n$ s.t.

$$\phi_k \text{ holds} \iff \exists \text{ counterexample of length } k$$

Then use SAT solver to check each formula starting with $k = 0$.

$\implies$ Finds *minimal* counterexamples, very quickly

BMC can be done in polynomial time

Bounded Model Checking (BMC)

Usage in NuSMV

Non-interactive mode: `NuSMV -bmc [-bmc_length n] <model>`

If no length is given, $n = 10$ is used as default.

Interactive mode: `go_bmc`, `check_ltlspec_bmc -p "formula" -k <bound>`

Can control shape of loop of counterexample (if applicable) with further parameters.

Bounded Model Checking (BMC)

Invariant Checking

NuSMV can check invariants using SAT solvers

`INVARSPEC <formula>` instead of `LTLSPEC` in files

Then use `-bmc` CLI flag as before to apply *2-step induction*.

Interactive mode: `check_invar_bmc -p "formula"`.

To use *complete* invariant checking (more powerful): `check_invar_bmc -a een-sorensson`

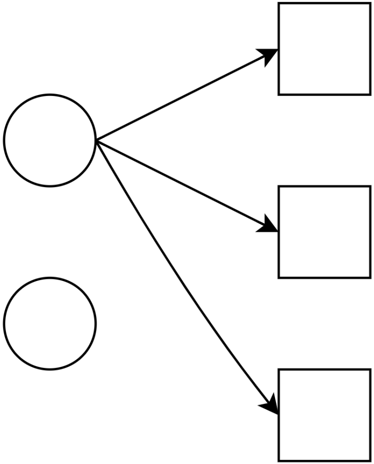
Case Study: Paxos

What is Paxos?

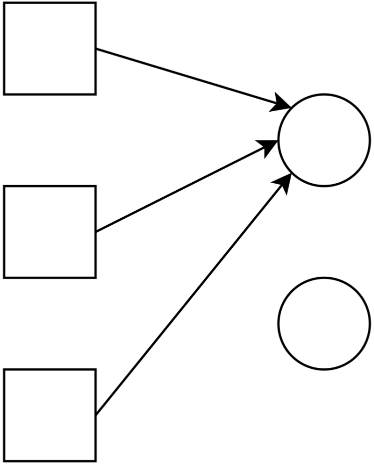
Paxos is a protocol that aims to achieve consensus on a value among a multitude of values proposed by three types of nodes:

- **Proposers** (○): propose a value on which they want the system to reach consensus. Every proposer has a *uniquely assigned set of rounds* during which they can send their proposals.
- **Acceptors** (□): they receive the values from the proposers and based on the rounds of the received messages they *deterministically* decide for which *value and round* to vote for, relaying the message to the learners.
- **Learners** (△): They receive the couple messages of `<vote, round>` and count them, if one couple has been voted *enough times* by the acceptors they will choose that value and spread it to the system.

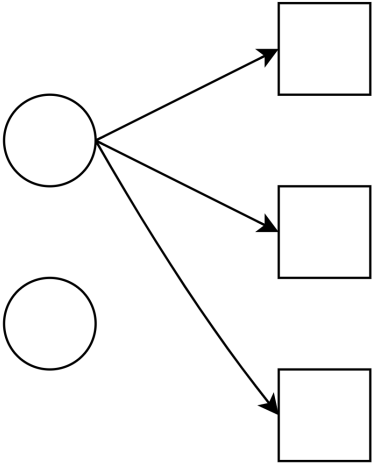
Prepare



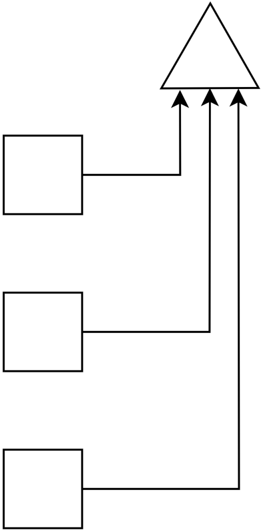
Promise



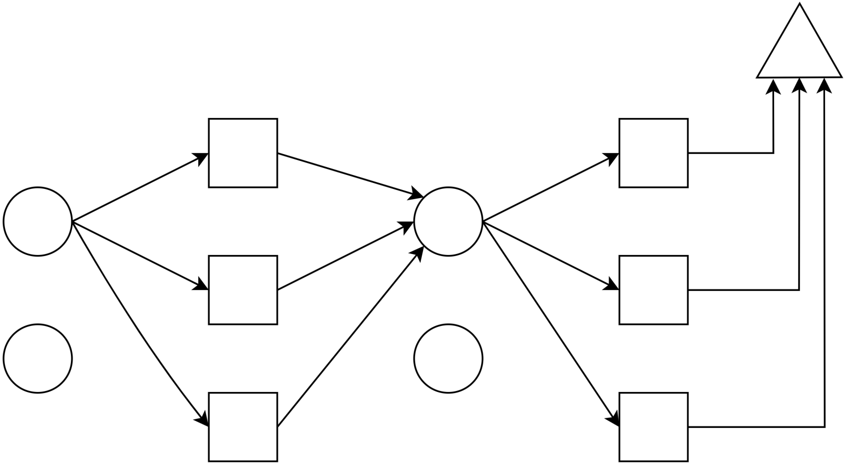
Accept



Learn



Paxos



Pseudocode

Algorithm 1 Paxos — Proposer p

```
Constants:
A, n, and f.        {A is the set of acceptors. n = |A| and f = ⌊(n − 1)/2⌋.}

Init:
crnd ← −1                                   {Current round number}

on ⟨PROPOSE, val⟩
    crnd ← pickNextRound(crnd)
    cval ← val
    P ← ∅
    send ⟨PREPARE, crnd⟩ to A

on ⟨PROMISE, rnd, vrnd, vval⟩ with rnd = crnd from acceptor a
    P ← P ∪ (vrnd, vval)

on event |P| ≥ n − f
    j = max{vrnd : (vrnd, vval) ∈ P}
    if j ≥ 0 then
        V = {vval : (j, vval) ∈ P}
        cval ← pick(V)        {Pick proposed value vval with largest vrnd}
    send ⟨ACCEPT, crnd, cval⟩ to A
```

Algorithm 2 Paxos — Acceptor a

```
Constants:
L                                           {Set of learners}

Init:
rnd ← −1
vrnd ← −1
vval ← −1

on ⟨PREPARE, prnd⟩ with prnd > rnd from proposer p
    rnd ← prnd
    send ⟨PROMISE, rnd, vrnd, vval⟩ to proposer p

on ⟨ACCEPT, i, v⟩ with i ≥ rnd from proposer p
    rnd ← i
    vrnd ← i
    vval ← v
    send ⟨LEARN, i, v⟩ to L
```

Algorithm 3 Paxos — Learner l

```
Init:
V ← ∅

on ⟨LEARN, (i, v)⟩ from acceptor a
    V ← V ⊎ (i, v)

on event ∃i, v : |{(i, v) : (i, v) ∈ V}| ≥ n − f
    v is chosen
```

Properties of Paxos

The Paxos Protocol has been formally proven to have the following properties:

- **CS1**: Only a proposed value may be chosen.
- **CS2**: Only a single value is chosen.
- **CS3**: Only a chosen value may be learned by a correct learner.
- **CS4**: If an acceptor has voted for value v at round i, then no value $v' \neq v$ can be chosen in any previous round.

The Properties are Verified

```
*** 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 > go
NuSMV > print_reachable_states
######################################################################
system diameter: 51
reachable states: 31028 (2^14.9213) out of 1.26214e+25 (2^83.3841)
######################################################################
NuSMV > process_model
The computation of reachable states has been completed.
The diameter of the FSM is 51.
-- specification  F ( G l1.decided)  is true
-- specification  G (((l1.decided & l1.chosen_value = 1) -> ( H p1.val = 1 |  H p2.val = 1)) & ((l1.decided & l1.chose
n_value = 2) -> ( H p1.val = 2 |  H p2.val = 2)))  is true
-- specification  G (((l1.v1_consensus -> ((l1.vote2 = l1.vote1 | !l1.v2_consensus) & (l1.vote3 = l1.vote1 | !l1.v3_co
nsensus))) & (l1.v2_consensus -> ((l1.vote2 = l1.vote1 | !l1.v1_consensus) & (l1.vote3 = l1.vote2 | !l1.v3_consensus))
)) & (l1.v3_consensus -> ((l1.vote2 = l1.vote3 | !l1.v2_consensus) & (l1.vote3 = l1.vote1 | !l1.v1_consensus))))  is t
rue
-- specification  G (l1.decided -> ( O p1.quorum |  O p2.quorum))  is true
-- specification  G (((((((l1.decided & l1.chosen_value = l1.vote1) & l1.vote1 = 1) ->  H (a1.last_voted_v = 1 | a1.l
ast_voted_v = -1)) & (((l1.decided & l1.chosen_value = l1.vote1) & l1.vote1 = 2) ->  H (a1.last_voted_v = 2 | a1.last_
voted_v = -1))) & (((l1.decided & l1.chosen_value = l1.vote2) & l1.vote2 = 1) ->  H (a2.last_voted_v = 1 | a2.last_vot
ed_v = -1))) & (((l1.decided & l1.chosen_value = l1.vote2) & l1.vote2 = 2) ->  H (a2.last_voted_v = 2 | a2.last_voted_
v = -1))) & (((l1.decided & l1.chosen_value = l1.vote3) & l1.vote3 = 1) ->  H (a3.last_voted_v = 1 | a3.last_voted_v =
 -1))) & (((l1.decided & l1.chosen_value = l1.vote3) & l1.vote3 = 2) ->  H (a3.last_voted_v = 2 | a3.last_voted_v = -1
)))  is true
NuSMV >
```

Code Generation

Code Generation for NuSMV

Translating Regular Expressions into Models

NuSMV language abstraction level awkward for large programs

Often repetitive structures

Can express complex semantics, however!

$\Longrightarrow$ Translate higher-level language to NuSMV

Toy Example: Regular Expressions

Regular Expressions: Syntax

Standard Unix Syntax

Bracketss:

$$regexp \rightarrow (regexp)$$

Alternation:

$$regexp \rightarrow regexp_1 | regexp_2$$

Concatenation:

$$regexp \rightarrow regexp_1\, regexp_2$$

Kleene Star:

$$regexp \rightarrow regexp^*$$

Thompson's Construction: Alternation

$$regexp \rightarrow s \mid t$$

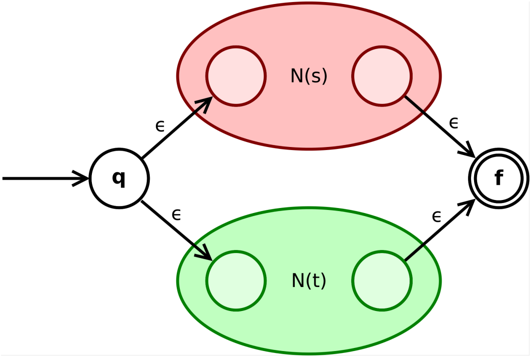


Figure: By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861675

Code Generation for NuSMV

Thompson's Construction: Concatenation

$regexp \rightarrow s\ t$

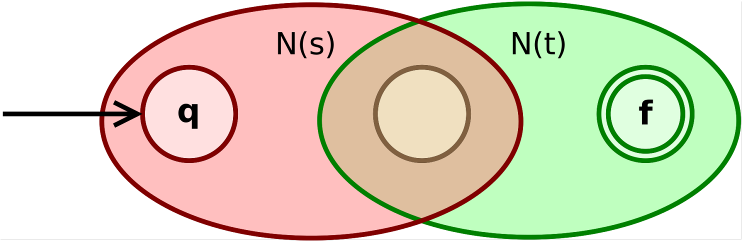


Figure: By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861673

Code Generation for NuSMV

Thompson's Construction: Kleene Star

$regexp \rightarrow s^*$

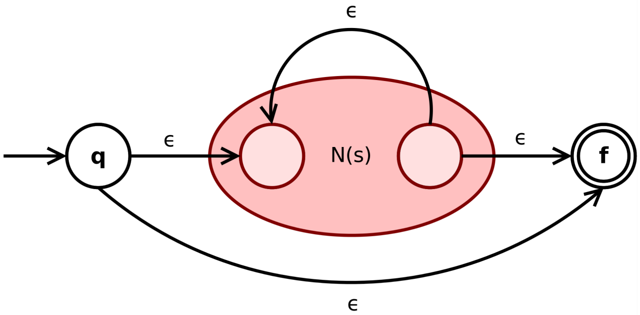


Figure: By Trapmoth - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21861674

Fin.