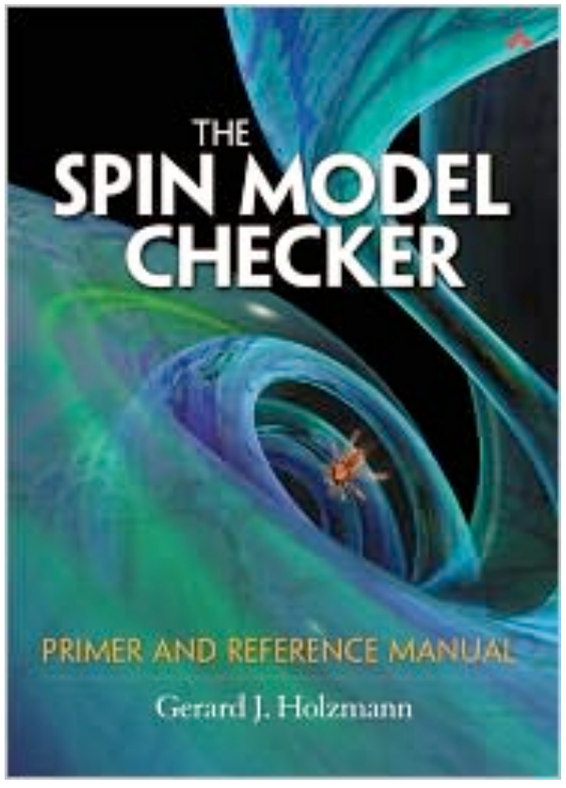
Sumofhellos mis-using mutexes

- Let's produce a version of sumofhellos that uses mutexes in an incorrect manner. (bad**`_sumofhellos.pml`**).
  - This is the advantage of a modelling language over a program language: We can say "make it so!" (to some extent, at least).
- As before, we can run simulations of this, and we observe failure.
- Again, we can build the state diagram and check every possible path through it, so determining the outcome of every possible interleaving
  - **`$> spin -run bad_sumofhellos.pml`**
- In this case we not only see an error indication, but a "trail" file has been created, which we can run with
  - **`$> spin -p -k bad_sumofhellos.pml.trail bad_sumofhellos.pml`**

Book Reference



The SPIN Model Checker: Primer and Reference Manual
By Gerald Holtzmann
Addison-Wesley, May 2011 ISBN 0321773713
$65 [Paperback]

The spin model checker : primer and reference manual /
Gerald J. Holzmann
Holzmann, Gerard J.

Printed Book | 2004

Available at Hamilton Lower (S-LEN 620 P496) plus 3 more see all

Click and Collect

Additional actions:

Modelling and validating 1553B p
model checker
Krishnan, Ranjani;Lalithambika, V R

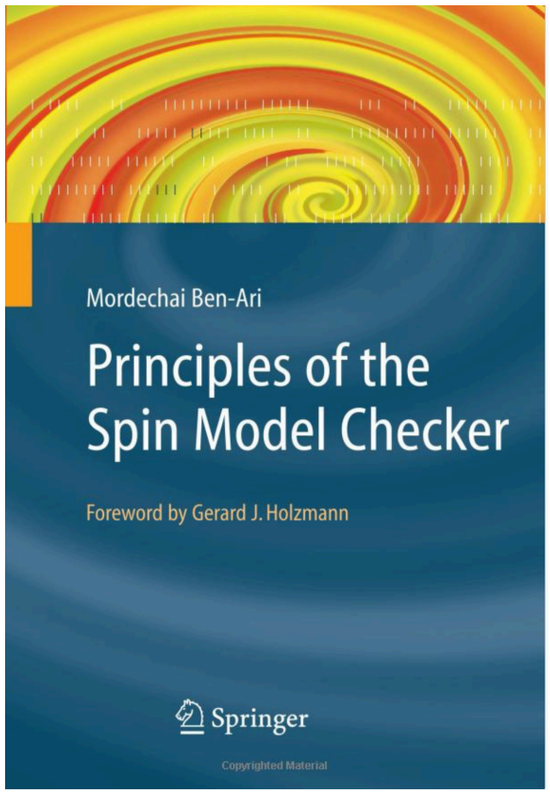
Conference | 2018 10th International C
Systems & Networks (COMSNETS) Commu
(COMSNETS), 2018 10th International Conf

TCD affiliated users click here for access

Items only show available close

Location (click for map)	Shelfmark	Status
Hamilton Lower	S-LEN 620 P496	IN
Hamilton Lower	S-LEN 620 P496;2	IN
Hamilton Lower	S-LEN 620 P496;3	IN
Hamilton Upper	620 P496;1	IN

Book Reference



Principles of the Spin Model Checker 2nd Edition
By Mordechai Ben-Ari
Springer, 2008 ISBN 978-1-84628-769-5
$63 [Paperback]

Principles of the Spin model checker / **Mordechai Ben-Ari**
Ben-Ari, M., 1948-

Printed Book | 2008

Not Available at Hamilton Lower (DUE 16-04-21) see all

Click and Collect

Additional actions:

→ Publisher description Click he
→ Table of contents only Click h

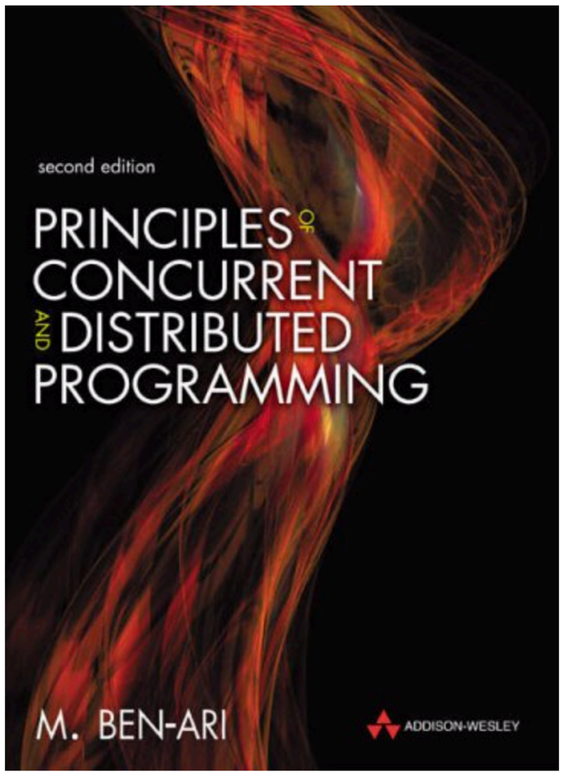
Blockchain based permis
control in Internet of Thi
Ali, Gauhar;Ahmad, Naveed;Cao,

Items

close

Location (click for map)	Shelfmark	Status
Hamilton Lower	S-LEN 500.164 P83;1	DUE 16-04-21
Hamilton Upper	500.164 P83	DUE 16-04-21

Book Reference



Principles of Concurrent and Distributed Programming
2nd Edition
By Mordechai Ben-Ari
Addison-Wesley, 2006 ISBN 978-0-321-31283-9
$120 [Paperback]

Principles of concurrent and distributed programming / **M. Ben-Ari**
Ben-Ari, M., 1948-

Printed Book | 2006

Available at Hamilton Lower (S-LEN 500.164 N093*1) plus 4+ more see all

Click and Collect

Additional actions:

Principles of concurrent and distribute
M. Ben-Ari
Ben-Ari, M., 1948-

Printed Book | 1990

Available at Santry Stacks (PL-157-209) see all

Items only show available cl

Location (click for map)	Shelfmark	Stat
Hamilton Lower	S-LEN 500.164 N093*1	IN
Hamilton Lower	S-LEN 500.164 N093*1;1	IN
Hamilton Lower	S-LEN 500.164 N093*1;2	IN
Hamilton Lower	S-LEN 500.164 N093*1;3	IN

Mutual Exclusion (without pthreads!)

- Early challenge in concurrency was to provide mutual exclusion
  - assuming that the only atomic operations were read and write
- First algorithm, for two processes only, described in 1965 by Edsger Dijkstra
  - He attributed it to T.J Dekker in 1962.
- It does not assume that two processes alternate access
    - It allow each process to return to the critical region as many times as it wants before the other one gets access.

Dekker's Algorithm (I)

```
bool turn;
bool flag[2];
byte count;

active [2] proctype mutex()
{    pid i, j;

    i = _pid;
    j = 1 - _pid;

again:

    < ACQUIRE "LOCK">

    count++;
    assert(count == 1); /* critical section */
    count--;

    <RELEASE "LOCK">

    goto again
}
```

- Overview:
  - variable **`count`** is the global shared variable
  - variable **`i`** refers to thread's own id, while **`j`** is that of the other thread.
- Body is an infinite loop
  - defined using label **`again:`**
  - and a **`goto`** statement !

Dekker's Algorithm (II)

```
flag[i] = true;
do
:: flag[j] ->
    if
    :: turn == j ->
        flag[i] = false;
        !(turn == j);
        flag[i] = true
    :: else
    fi
:: else ->
    break
od;
```

- Acquiring the lock:
  - set own **`flag`** to say "going in!"
  - while other **`flag`** is true:
    - if other's **`turn`**:
      turn own **`flag`** off, wait for turn, turn **`flag`** on again
- When we exit, the other's **`flag`** is off and it is our **`turn`**.

("The Spin Model-Checker", p20)

Dekker's Algorithm (III)

```
turn = j;
flag[i] = false;
```

- Releasing the lock:
  - It is their **`turn`**
  - Our **`flag`** is now false.

("The Spin Model-Checker", p20)

Checking Dekker

- We can run it: `spin -u1000 dekker.pml`
- We can verify it `spin -a dekker.pml ; cc -o pan pan.c ; ./pan`
- ????

- SPIN takes a Promela model and compiles it into a C program that builds and analyses that model.
- The executable is usually called "pan"
  - short for "Protocol Analyser" - the first application area for Promela/SPIN
- splitting **`spin -run model.pml`** into **`spin -a model.pml`** and then using **`pan`** allows a lot of flexibility in the kinds of analyses that can be done.

Improvement (Doran&Thomas 1980)

```
flag[i] = true;
if
:: flag[j] ->
    if
    :: turn == j ->
        flag[i] = false;
        !(turn == j);
        flag[i] = true
    :: else
    fi;
    (!flag[j]);
:: else ->
fi;
```

- Is top-level loop necessary?
  - Hard to say without careful analysis
- Acquiring the lock:
  - set own **`flag`** to say "going in!"
  - if other **`flag`** is true:
    - if other's **`turn`**:
      turn own **`flag`** off, wait for **`turn`**, turn **`flag`** on again
    - then, wait for their **`flag`** to turn off.
      - Not in SPIN/Promela book !
- When we exit, the other's **`flag`** is off and it is our **`turn`**.

("The Spin Model-Checker", p22, plus note in Errata)

```
byte count;
byte x, y, z;
active [2] proctype user()
{ byte me = _pid + 1; /* me is 1 or 2 */
L1: x = me;
L2: if
  :: (y != 0 && y != me) -> goto L1
  :: (y == 0 || y == me)
  fi;
L3: z = me;
L4: if
  :: (x != me)  -> goto L1
  :: (x == me)
  fi;
L5: y = me;
L6: if
  :: (z != me) -> goto L1
  :: (z == me)
  fi;

L7: progress:/* success: enter critical section */
  count++;
  assert(count == 1);
  count--;
  goto L1
}
```

Something Broken

- A major computer manufacturer recommended this....
  - SPIN very quickly finds a counter-example!

```
spin -run mutex_flaw.pml
```

Shrinking SPIN's counter-example

- Failure is noted and a .trail file created
  - the "depth" is how many steps were taken to failure
  - We can "run" the trail: **`spin -p -t mutex_flaw.pml`**

- Two ways to find a shorter counter-example
  - Keep track of depth and iteratively search for minimal depth
    `cc -DREACH -o pan pan.c ; ./pan -i -mDD` where DD is known depth.
  - Use breadth-first search instead (default is depth-first)
    `cc -DBFS -o pan pan.c ; ./pan`

The Closed World Assumption

- A Model Checker seeks to establish the truth of something by trying exhaustively and failing to demonstrate the ***negation*** of it.
  - e.g. to show that mutex holds, it tries to demonstrate (by counterexample) that it doesn't hold. If it fails to do so, it is taken as proof that the mutex holds.
- This only corresponds to what we normally understand to be "proof" if it is assumed that the model checker knows everything (i.e. can prove everything provable) about the system -- it's called the "Closed World Assumption."
- A property deemed true, or discovered to be false, by a model-checker, may not be so in the real world.
  - This typically arises because a model only covers some aspects of a systems behaviour, and other aspects, not modelled, may also have an impact on behaviour.

Promela Verification Constructs

- So far, we have seen that SPIN does some standard verification checks by default
  - e.g. Deadlock Freedom
- There are a number of other common cases that can be handled without using Linear Temporal Logic (LTL)
  - Basic Assertions
  - Special Labels:
    - End-States
    - Progress-States

Promela – *Basic Assertion*

- Format: assert(<expression>)
  - where `<expression>` must evaluate to `true` or
    - simulation will be aborted or
    - verification will fail

      Note: an `assert` is evaluated during both simulation and verification.
- Already seen its use in **`sumofhellos.pml`**, for example

Promela Labels

- Promela allows labels to be attached to statements
  - (e.g. see **`dekker.pml`** or **`doran.pml`**)
- By using labels with a specific ***prefix***, we can trigger certain kinds of correctness checks.

Promela – *End-State Label*

- Promela always verifies that no deadlock occurs.
- It assumes that the only valid end-states for a system are where:
  - each process is the end of its code.
- If it can show an end state where a process is not at the end of its code, it considers that an error (Deadlock).
- Sometimes, it is legitimate for a process not to end up at the end of its code. You can label such states **`end`**... to indicate to Promela that they are valid end points.

  `end`, **`end_one`**, **`end00`** – anything starting with **`end`**.

end...: label example

- An example of a valid-end state, that is not the end of a program, is where a program that is not meant to terminate (i.e. a web-server) is waiting for client requests to arrive.
- Consider an example of two servers talking to a client that accesses them both and terminates [SMC-programs/ch04/end.pml]
  - When the client terminates, both servers are left waiting on a request
  - SPIN reports an error - but this is erroneous

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  byte request = 0;
5
6  active proctype Server1() {
7    do
8    :: request == 1 ->
9            printf("Service 1\n");
10           request = 0;
11   od
12 }
13
14 active proctype Server2() {
15   do
16   :: request == 2 ->
17           printf("Service 2\n");
18           request = 0;
19   od
20 }
21
22 active proctype Client() {
23   request = 1;
24   request == 0;
25   request = 2;
26   request == 0;
27 }
```

`end...:` label example

- We fix this by labelling the (start of) the **`do .. od`** statement in each server.
- SPIN is now happy

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  byte request = 0;
5
6  active proctype Server1() {
7  end1: do
8        :: request == 1 ->
9                printf("Service 1\n");
10               request = 0;
11       od
12 }
13
14 active proctype Server2() {
15 end2: do
16       :: request == 2 ->
17               printf("Service 2\n");
18               request = 0;
19       od
20 }
21
22 active proctype Client() {
23   request = 1;
24   request == 0;
25   request = 2;
26   request == 0;
27 }
```

Promela – *Progress-State Label*

- A system can have loops – cycles of sequences of states passed through infinitely often.
  - The question is, are such loops desirable or not;
    - if progress is made each time, then they are desirable loops -- progress cycles;
    - otherwise, they are undesirable *non-progress cycles*, where the system is doing something but not progressing.
- A given loop might never get executed
  - This can happen if there is always something else that can run elsewhere, and the "scheduler" is being "unkind" to this loop
  - If the loop contains a state that should be happening repeatedly, then this is a case of ***starvation***.
- You can label such states **`progress`**... to indicate to Promela that they should occur in any (infinite) loop.

  `progress`, **`progress_one`**, **`progress00`** – anything starting with **`progress`**.

`progress...:` label example

- Here we another implementation of mutual exclusion
  - It is deadlock-free and only one thread is in critical region at any time.
  - A standard SPIN verification run shows no problems
- However, can one thread run at the expense of the other?

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  bool wantP = false, wantQ = false;
5
6  active proctype P() {
7    do
8    :: wantP = true;
9       do
10      :: wantQ -> wantP = false; wantP = true
11      :: else -> break
12      od;
13      // critical section
14      wantP = false
15   od
16 }
17
18 active proctype Q() {
19   do
20   :: wantQ = true;
21      do
22      :: wantP -> wantQ = false; wantQ = true
23      :: else -> break
24      od;
25      // critical section
26      wantQ = false
27   od
28 }
```

`progress...:` label example

- We can use a progress label
  - We label the critical section in **`P()`**
- We need a special verifier:
  - Optimised for "non-progress" (NP) cycles

    `cc -DNP -o pan pan.c`
- We need to ask it to find such cycles

  `./pan -l`

```
1  /* Copyright 2007 by Moti Ben-Ari
2  under the GNU GPL; see readme.txt */
3
4  bool wantP = false, wantQ = false;
5
6  active proctype P() {
7    do
8    :: wantP = true;
9       do
10      :: wantQ -> wantP = false; wantP = true
11      :: else -> break
12      od;
13      progressP: // critical section
14      wantP = false
15   od
16 }
17
18 active proctype Q() {
19   do
20   :: wantQ = true;
21      do
22      :: wantP -> wantQ = false; wantQ = true
23      :: else -> break
24      od;
25      // critical section
26      wantQ = false
27   od
28 }
```

Showing infinite cycles

- SPIN has a way of showing infinite cycles

```
:- spin -t -p SMC-5-1.pml
starting claim 2
spin: couldn't find claim 2 (ignored)
  2:    proc  1 (Q:1) SMC-5-1.pml:21 (state 1)  [wantQ = 1]
  <<<<<START OF CYCLE>>>>>
  4:    proc  1 (Q:1) SMC-5-1.pml:24 (state 5)  [else]
  6:    proc  1 (Q:1) SMC-5-1.pml:27 (state 10) [wantQ = 0]
  8:    proc  1 (Q:1) SMC-5-1.pml:21 (state 1)  [wantQ = 1]
spin: trail ends after 8 steps
#processes: 2
                wantP = 0
                wantQ = 1
  8:    proc  1 (Q:1) SMC-5-1.pml:22 (state 7)
  8:    proc  0 (P:1) SMC-5-1.pml:7 (state 11)
2 processes created
```

- it marks the start of the cycle with **<<<<<START OF CYCLE>>>>>**
- the last state shown is the same as the one just before that marker

"never claims" ???

- When we run the liveness analysis we see that the SPIN output is suddenly all about "claims" and "never claims".

```
Full statespace search for:
        never claim             + (:np_:)
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
```

- What are these?

Never Claims

- Never Claims are the mechanism used by SPIN to check LTL properties
- The state models produced by SPIN are not just simple graphs
  - They are automata, known as Büchi automata
- Büchi Automata:
  - Finite state machines
  - With criteria defined for accepting both finite and infinite languages
  - This means that accepting states can be state-loops
- LTL properties can be converted to Büchi automata
  - These can be expressed in Promela
  - They run in parallel with the Promela model defined in a `.pml` file.

Finite State Machines/Automata (FSM/FSA)

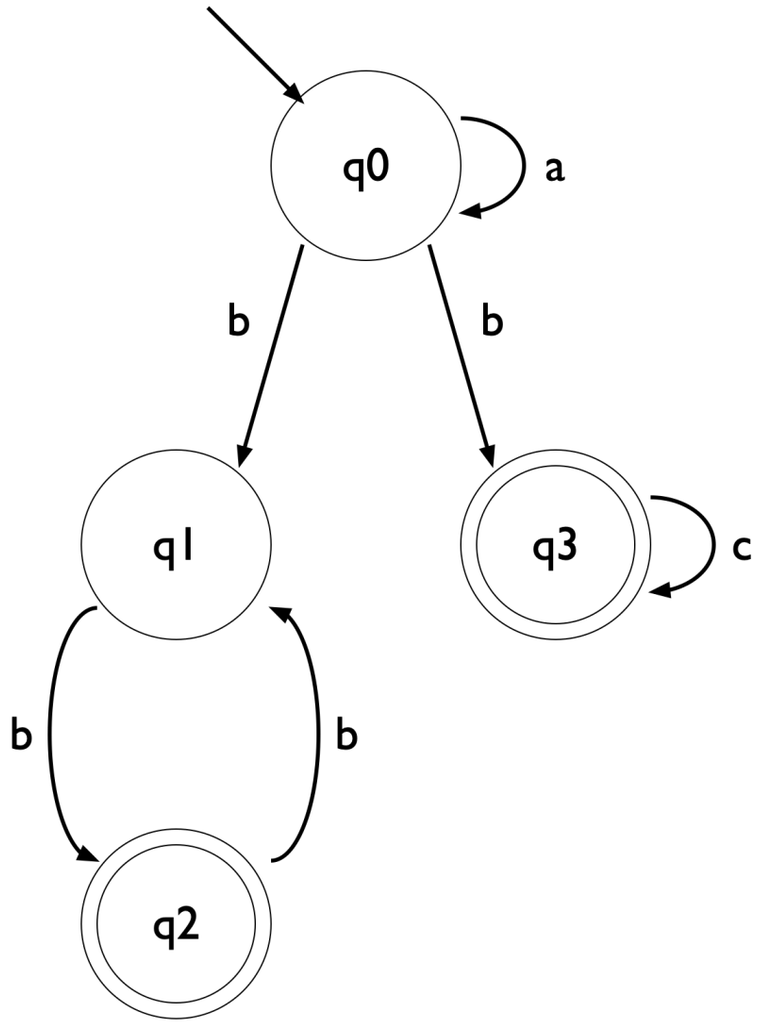
- Composed of a finite number of states
  - One is designated as the starting state
  - One, or more, are designated as accepting (final) states
- Responds to events
  - Each state responds to some of the events
  - A response "consumes" the event, and may also change the state it is in (moving to "next" state).
- Non-determinism
  - A state responding to an event may have a choice between different states to which it can change.
    - If all states have no such choice then we have a Deterministic Finite Automata (DFA)
    - If any state has such a choice, then we have a Non-deterministic Finite Automata (NDFA)

FSAs "accept" event-sequences

- Given a (N)DFA and a sequence of events, we can ask
  - Can the (N)DFA perform this sequence?
  - Does the starting event "respond" to the first event?
    - If so, does the next state "respond" to the second event?
    - and so on.
- We say that a (N)DFA "accepts" a sequence of events if there is a way to respond to all of the events such that it ends up in one of its "accepting" states.

NDFA Example

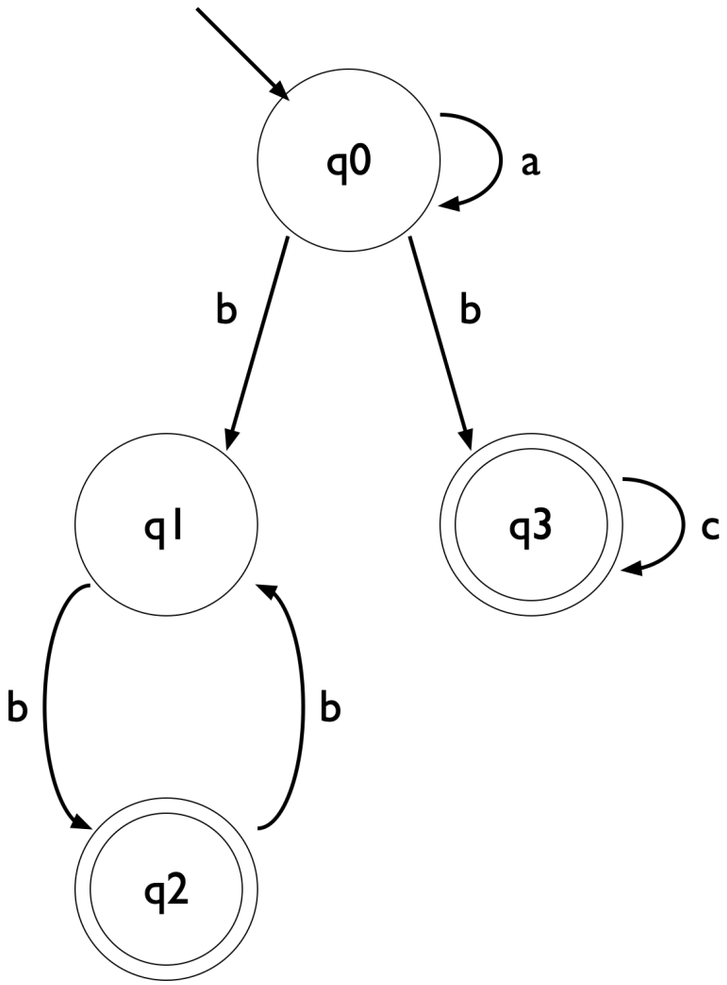
- Regular expressions are a shorthand way to describe finite sequences
  - `seq1 seq2`
    - seq1 followed by seq2
  - `seq1 + seq2`
    - either seq1 or seq2
  - `seq*`
    - zero or more repetitions of seq
  - `seq+`
    - one or more repetitions of seq
- E.g.,: `a*((bb)+ + bc*)`
  - zero or more as,
    followed by either:
    an even non-zero number of bs;
    or one b followed by zero or more cs.



Principles of the Spin Model Checker, M. Ben-Ari, Sec 8.1, pp125-7

NDFA in Promela

Any NDFA can be written in Promela




```
1  /* Copyright 2007 by Moti Ben-Ari
2     under the GNU GPL; see readme.txt */
3
4  #define LEN 5
5
6  active proctype FA() {
7    byte h;
8    byte i[LEN];
9    i[0] = 'a'; i[1] = 'a'; i[2] = 'b'; i[3] = 'b'; i[4] = '.';
10 q0: if
11     :: i[h] == 'a'  -> h++; goto q0
12     :: i[h] == 'b'  -> h++; goto q3
13     :: i[h] == 'b'  -> h++; goto q1
14     fi;
15 q1: if
16     :: i[h] == 'b'  -> h++; goto q2
17     fi;
18 q2: if
19     :: i[h] == 'b'  -> h++; goto q1
20     :: i[h] == '.'  -> goto accept
21     fi;
22 q3: if
23     :: i[h] == 'c'  -> h++; goto q3
24     :: i[h] == '.'  -> goto accept
25     fi;
26 accept:
27     printf("Accepted!\n");
28     // assert(false)
29 }
```

Applying SPIN to NDFA

- We can simulate the example,
  - **`spin SMC-8-1.pml`**

    but because the first `b` results in either `q1` or `q3` being chosen non-deterministically, we are not guaranteed to get the accepting outcome
- We can verify it
  - spin -run SMC-8-1.pml
    Here we have only 1 process, so we get a "timeout" rather than any deadlock - not helpful.
- We can "falsify" it.
  - We want to see if we can get to **`Accepted!`** message, so **`assert(false)`** just afterwards !?!?!
  - A verification run will find the only "failure" in the system (**`false`**), and generate the sequence of steps leading through label "**`accept:`**"

The “flip side” of counter-examples

- SPIN is designed to exhaustively check a model to see if any desired property is in fact false.
  - If it finds such a failure, it generates a counter-example - sequence of moves leading to failure.
    If it doesn’t, it reports that all is well.
- We can use SPIN to find solutions to problems too.
  - We design our model so that:
    - (a) there is a point in the code that corresponds to having a solution, or
      (b) we specify a property that is true when a solution is found.
  - We then:
    - (a) add in an **`assert(false)`** after the solution point in the code, or
      (b) specify the negation of the property
  - SPIN will then, eventually, find the **`assert(false)`**, or where the negated property is false, and report this as a failure with a “counter-example” that leads to that point - the solution !