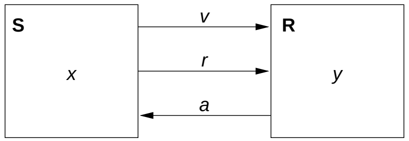
Verifying Concurrent Systems

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

1. **Verification by Computer-Supported Proving**
2. The Model Checker Spin
3. Verification by Automatic Model Checking

A Bit Transmission Protocol



var x, y
var $v := 0, r := 0, a := 0$

S: **loop**
  choose $x \in \{0,1\}$
  $1 : v, r := x, 1$
  $2 :$ **wait** $a = 1$
    $r := 0$
  $3 :$ **wait** $a = 0$

||

R: **loop**
  $1 :$ **wait** $r = 1$
    $y, a := v, 1$
  $2 :$ **wait** $r = 0$
    $a := 0$

Transmit a sequence of bits through a wire.

A (Simplified) Model of the Protocol

$State := PC^2 \times (\mathbb{N}_2)^5$

$I(p,q,x,y,v,r,a) :\Leftrightarrow p = q = 1 \wedge x \in \mathbb{N}_2 \wedge v = r = a = 0.$
$R(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad S1(\ldots) \vee S2(\ldots) \vee S3(\ldots) \vee R1(\ldots) \vee R2(\ldots).$

$S1(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad p = 1 \wedge p' = 2 \wedge v' = x \wedge r' = 1 \wedge$
$\quad q' = q \wedge x' = x \wedge y' = y \wedge v' = v \wedge a' = a.$
$S2(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad p = 2 \wedge p' = 3 \wedge a = 1 \wedge r' = 0 \wedge$
$\quad q' = q \wedge x' = x \wedge y' = y \wedge v' = v \wedge a' = a.$
$S3(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad p = 3 \wedge p' = 1 \wedge a = 0 \wedge x' \in \mathbb{N}_2 \wedge$
$\quad q' = q \wedge y' = y \wedge v' = v \wedge r' = r \wedge a' = a.$
$R1(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad q = 1 \wedge q' = 2 \wedge r = 1 \wedge y' = v \wedge a' = 1 \wedge$
$\quad p' = p \wedge x' = x \wedge v' = v \wedge r' = r.$
$R2(\langle p,q,x,y,v,r,a\rangle, \langle p',q',x',y',v',r',a'\rangle) :\Leftrightarrow$
$\quad q = 2 \wedge q' = 1 \wedge r = 0 \wedge a' = 0 \wedge$
$\quad p' = p \wedge x' = x \wedge y' = y \wedge v' = v \wedge r' = r.$

A Verification Task

$\langle I, R \rangle \models \Box(q = 2 \Rightarrow y = x)$

$Invariant(p, \ldots) \Rightarrow (q = 2 \Rightarrow y = x)$

$I(p, \ldots) \Rightarrow Invariant(p, \ldots)$
$R(\langle p, \ldots \rangle, \langle p', \ldots \rangle) \wedge Invariant(p, \ldots) \Rightarrow Invariant(p', \ldots)$

$$
\begin{aligned}
& Invariant(p, q, x, y, v, r, a) :\Leftrightarrow \\
& \quad (p = 1 \vee p = 2 \vee p = 3) \wedge (q = 1 \vee q = 2) \wedge \\
& \quad (x = 0 \vee x = 1) \wedge (v = 0 \vee v = 1) \wedge (r = 0 \vee r = 1) \wedge (a = 0 \vee a = 1) \wedge \\
& \quad (p = 1 \Rightarrow q = 1 \wedge r = 0 \wedge a = 0) \wedge \\
& \quad (p = 2 \Rightarrow r = 1) \wedge \\
& \quad (p = 3 \Rightarrow r = 0) \wedge \\
& \quad (q = 1 \Rightarrow a = 0) \wedge \\
& \quad (q = 2 \Rightarrow (p = 2 \vee p = 3) \wedge a = 1 \wedge y = x) \wedge \\
& \quad (r = 1 \Rightarrow p = 2 \wedge v = x)
\end{aligned}
$$

The invariant captures the essence of the protocol.

The RISC ProofNavigator Theory

```
newcontext "protocol";

p: NAT; q: NAT; x: NAT; y: NAT; v: NAT; r: NAT; a: NAT;
p0: NAT; q0: NAT; x0: NAT; y0: NAT; v0: NAT; r0: NAT; a0: NAT;

S1: BOOLEAN =
  p = 1 AND p0 = 2 AND v0 = x AND r0 = 1 AND
  q0 = q AND x0 = x AND y0 = y AND v0 = v AND a0 = a;
S2: BOOLEAN =
  p = 2 AND p0 = 3 AND a = 1 AND r0 = 0 AND
  q0 = q AND x0 = x AND y0 = y AND v0 = v AND a0 = a;
S3: BOOLEAN =
  p = 3 AND p0 = 1 AND a = 0 AND (x0 = 0 OR x0 = 1) AND
  q0 = q AND y0 = y AND v0 = v AND r0 = r AND a0 = a;

R1: BOOLEAN =
  q = 1 AND q0 = 2 AND r = 1 AND y0 = v AND a0 = 1 AND
  p0 = p AND x0 = x AND v0 = v AND r0 = r;
R2: BOOLEAN =
  q = 2 AND q0 = 1 AND r = 0 AND a0 = 0 AND
  p0 = p AND x0 = x AND y0 = y AND v0 = v AND r0 = r;
```


The RISC ProofNavigator Theory

```
Init: BOOLEAN =
  p = 1 AND q = 1 AND (x = 0 OR x = 1) AND
  v = 0 AND r = 0 AND a = 0;

Step: BOOLEAN =
  S1 OR S2 OR S3 OR R1 OR R2;

Invariant: (NAT, NAT, NAT, NAT, NAT, NAT, NAT)->BOOLEAN =
  LAMBDA(p, q, x, y, v, r, a: NAT):
    (p = 1 OR p = 2 OR p = 3) AND
    (q = 1 OR q = 2) AND
    (x = 0 OR x = 1) AND
    (v = 0 OR v = 1) AND
    (r = 0 OR r = 1) AND
    (a = 0 OR a = 1) AND
    (p = 1 => q = 1 AND r = 0 AND a = 0) AND
    (p = 2 => r = 1) AND
    (p = 3 => r = 0) AND
    (q = 1 => a = 0) AND
    (q = 2 => (p = 2 OR p = 3) AND a = 1 AND y = x) AND
    (r = 1 => p = 2 AND v = x);
```


The RISC ProofNavigator Theory

```
Property: BOOLEAN =
  q = 2 => y = x;

VC0: FORMULA
  Invariant(p, q, x, y, v, r, a) => Property;

VC1: FORMULA
  Init => Invariant(p, q, x, y, v, r, a);

VC2: FORMULA
  Step AND Invariant(p, q, x, y, v, r, a) =>
    Invariant(p0, q0, x0, y0, v0, r0, a0);
```

The Proofs

```
[vd2]: expand Invariant, Property in m2v
  [rle]: proved (CVCL)

[wd2]: expand Init, Invariant in nra
  [ipl]: proved(CVCL)

[xd2]: expand Step, Invariant, S1, S2, S3, R1, R2
  [6ss]: proved(CVCL)
```

More instructive: proof attempts with wrong or too weak invariants (see demonstration).

A Client/Server System

Client system $C_i = \langle IC_i, RC_i \rangle$.

$State := PC \times \mathbb{N}_2 \times \mathbb{N}_2$.
$Int := \{R_i, S_i, C_i\}$.

$IC_i(pc, request, answer) :\Leftrightarrow$
$\quad pc = R \wedge request = 0 \wedge answer = 0.$
$RC_i(I, \langle pc, request, answer \rangle,$
$\quad \langle pc', request', answer' \rangle) :\Leftrightarrow$
$\quad (I = R_i \wedge pc = R \wedge request = 0 \wedge$
$\quad\quad pc' = S \wedge request' = 1 \wedge answer' = answer) \vee$
$\quad (I = S_i \wedge pc = S \wedge answer \neq 0 \wedge$
$\quad\quad pc' = C \wedge request' = request \wedge answer' = 0) \vee$
$\quad (I = C_i \wedge pc = C \wedge request = 0 \wedge$
$\quad\quad pc' = R \wedge request' = 1 \wedge answer' = answer) \vee$

$\quad (I = \overline{REQ}_i \wedge request \neq 0 \wedge$
$\quad\quad pc' = pc \wedge request' = 0 \wedge answer' = answer) \vee$
$\quad (I = ANS_i \wedge$
$\quad\quad pc' = pc \wedge request' = request \wedge answer' = 1).$

```
Client(ident):
  param ident
begin
  loop
    ...
 R: sendRequest()
 S: receiveAnswer()
 C: // critical region
    ...
    sendRequest()
  endloop
end Client
```


A Client/Server System (Contd)

Server system $S = \langle IS, RS \rangle$.

$State := (\mathbb{N}_3)^3 \times (\{1,2\} \rightarrow \mathbb{N}_2)^2$.
$Int := \{D1, D2, F, A1, A2, W\}$.

$IS(given, waiting, sender, rbuffer, sbuffer) :\Leftrightarrow$
$\quad given = waiting = sender = 0 \wedge$
$\quad rbuffer(1) = rbuffer(2) = sbuffer(1) = sbuffer(2) = 0.$

$RS(I, \langle given, waiting, sender, rbuffer, sbuffer \rangle,$
$\quad \langle given', waiting', sender', rbuffer', sbuffer' \rangle) :\Leftrightarrow$
$\quad \exists i \in \{1,2\}:$
$\quad\quad (I = D_i \wedge sender = 0 \wedge rbuffer(i) \neq 0 \wedge$
$\quad\quad sender' = i \wedge rbuffer'(i) = 0 \wedge$
$\quad\quad U(given, waiting, sbuffer) \wedge$
$\quad\quad \forall j \in \{1,2\} \backslash \{i\}: U_j(rbuffer)) \vee$
$\quad \ldots$

$U(x_1, \ldots, x_n) :\Leftrightarrow x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n.$
$U_j(x_1, \ldots, x_n) :\Leftrightarrow x'_1(j) = x_1(j) \wedge \ldots \wedge x'_n(j) = x_n(j).$

```
Server:
  local given, waiting, sender
begin
  given := 0; waiting := 0
  loop
D:  sender := receiveRequest()
    if sender = given then
      if waiting = 0 then
F:      given := 0
      else
A1:     given := waiting;
        waiting := 0
        sendAnswer(given)
      endif
    elsif given = 0 then
A2:   given := sender
      sendAnswer(given)
    else
W:    waiting := sender
    endif
  endloop
end Server
```


A Client/Server System (Contd'2)

$\ldots$
$(I = F \wedge sender \neq 0 \wedge sender = given \wedge waiting = 0 \wedge$
$\quad given' = 0 \wedge sender' = 0 \wedge$
$\quad U(waiting, rbuffer, sbuffer)) \vee$

$(I = A1 \wedge sender \neq 0 \wedge sbuffer(waiting) = 0 \wedge$
$\quad sender = given \wedge waiting \neq 0 \wedge$
$\quad given' = waiting \wedge waiting' = 0 \wedge$
$\quad sbuffer'(waiting) = 1 \wedge sender' = 0 \wedge$
$\quad U(rbuffer) \wedge$
$\quad \forall j \in \{1,2\} \backslash \{waiting\}: U_j(sbuffer)) \vee$

$(I = A2 \wedge sender \neq 0 \wedge sbuffer(sender) = 0 \wedge$
$\quad sender \neq given \wedge given = 0 \wedge$
$\quad given' = sender \wedge$
$\quad sbuffer'(sender) = 1 \wedge sender' = 0 \wedge$
$\quad U(waiting, rbuffer) \wedge$
$\quad \forall j \in \{1,2\} \backslash \{sender\}: U_j(sbuffer)) \vee$
$\ldots$

```
Server:
  local given, waiting, sender
begin
  given := 0; waiting := 0
  loop
D:  sender := receiveRequest()
    if sender = given then
      if waiting = 0 then
F:      given := 0
      else
A1:     given := waiting;
        waiting := 0
        sendAnswer(given)
      endif
    elsif given = 0 then
A2:   given := sender
      sendAnswer(given)
    else
W:    waiting := sender
    endif
  endloop
end Server
```

A Client/Server System (Contd'3)

$\ldots$

$(I = W \wedge sender \neq 0 \wedge sender \neq given \wedge given \neq 0 \wedge$
$\quad waiting' := sender \wedge sender' = 0 \wedge$
$\quad U(given, rbuffer, sbuffer)) \vee$

$\exists i \in \{1,2\}:$

$(I = REQ_i \wedge rbuffer'(i) = 1 \wedge$
$\quad U(given, waiting, sender, sbuffer) \wedge$
$\quad \forall j \in \{1,2\} \backslash \{i\} : U_j(rbuffer)) \vee$

$(I = \overline{ANS_i} \wedge sbuffer(i) \neq 0 \wedge$
$\quad sbuffer'(i) = 0 \wedge$
$\quad U(given, waiting, sender, rbuffer) \wedge$
$\quad \forall j \in \{1,2\} \backslash \{i\} : U_j(sbuffer)).$

```
Server:
  local given, waiting, sender
begin
  given := 0; waiting := 0
  loop
D:   sender := receiveRequest()
     if sender = given then
       if waiting = 0 then
F:       given := 0
       else
A1:      given := waiting;
         waiting := 0
         sendAnswer(given)
       endif
     elsif given = 0 then
A2:    given := sender
       sendAnswer(given)
     else
W:     waiting := sender
     endif
  endloop
end Server
```


A Client/Server System (Contd'4)

$State := (\{1,2\} \rightarrow PC) \times (\{1,2\} \rightarrow \mathbb{N}_2)^2 \times (\mathbb{N}_3)^2 \times (\{1,2\} \rightarrow \mathbb{N}_2)^2$

$I(pc, request, answer, given, waiting, sender, rbuffer, sbuffer) :\Leftrightarrow$
$\quad \forall i \in \{1,2\} : IC(pc_i, request_i, answer_i) \wedge$
$\quad IS(given, waiting, sender, rbuffer, sbuffer)$

$R(\langle pc, request, answer, given, waiting, sender, rbuffer, sbuffer \rangle,$
$\quad \langle pc', request', answer', given', waiting', sender', rbuffer', sbuffer' \rangle) :\Leftrightarrow$
$\quad (\exists i \in \{1,2\} : RC_{local}(\langle pc_i, request_i, answer_i \rangle, \langle pc'_i, request'_i, answer'_i \rangle) \wedge$
$\qquad \langle given, waiting, sender, rbuffer, sbuffer \rangle =$
$\qquad\quad \langle given', waiting', sender', rbuffer', sbuffer' \rangle) \vee$
$\quad (RS_{local}(\langle given, waiting, sender, rbuffer, sbuffer \rangle,$
$\qquad\quad \langle given', waiting', sender', rbuffer', sbuffer' \rangle) \wedge$
$\qquad \forall i \in \{1,2\} : \langle pc_i, request_i, answer_i \rangle = \langle pc'_i, request'_i, answer'_i \rangle) \vee$
$\quad (\exists i \in \{1,2\} : External(i, \langle request_i, answer_i, rbuffer, sbuffer \rangle,$
$\qquad\qquad \langle request'_i, answer'_i, rbuffer', sbuffer' \rangle) \wedge$
$\qquad pc = pc' \wedge \langle sender, waiting, given \rangle = \langle sender', waiting', given' \rangle)$

The Verification Task

$\langle I, R \rangle \models \Box\neg(pc_1 = C \wedge pc_2 = C)$

$Invariant(pc, request, answer, sender, given, waiting, rbuffer, sbuffer) :\Leftrightarrow$
$\quad \forall i \in \{1,2\}:$
$\quad (pc(i) = C \vee sbuffer(i) = 1 \vee answer(i) = 1 \Rightarrow$
$\qquad given = i \wedge$
$\qquad \forall j : j \neq i \Rightarrow pc(j) \neq C \wedge sbuffer(j) = 0 \wedge answer(j) = 0) \wedge$
$\quad (pc(i) = R \Rightarrow$
$\qquad sbuffer(i) = 0 \wedge answer(i) = 0 \wedge$
$\qquad (i = given \Leftrightarrow request(i) = 1 \vee rbuffer(i) = 1 \vee sender = i) \wedge$
$\qquad (request(i) = 0 \vee rbuffer(i) = 0)) \wedge$
$\quad (pc(i) = S \Rightarrow$
$\qquad (sbuffer(i) = 1 \vee answer(i) = 1 \Rightarrow$
$\qquad\quad request(i) = 0 \wedge rbuffer(i) = 0 \wedge sender \neq i) \wedge$
$\qquad (i \neq given \Rightarrow$
$\qquad\quad request(i) = 0 \vee rbuffer(i) = 0)) \wedge$
$\quad (pc(i) = C \Rightarrow$
$\qquad request(i) = 0 \wedge rbuffer(i) = 0 \wedge sender \neq i \wedge$
$\qquad sbuffer(i) = 0 \wedge answer(i) = 0) \wedge$
$\quad \ldots$

The Verification Task (Contd)

$\ldots$
$(sender = 0 \wedge (request(i) = 1 \vee rbuffer(i) = 1) \Rightarrow$
$\quad sbuffer(i) = 0 \wedge answer(i) = 0) \wedge$
$(sender = i \Rightarrow$
$\quad (waiting \neq i) \wedge$
$\quad (sender = given \wedge pc(i) = R \Rightarrow$
$\qquad request(i) = 0 \wedge rbuffer(i) = 0) \wedge$
$\quad (pc(i) = S \wedge i \neq given \Rightarrow$
$\qquad request(i) = 0 \wedge rbuffer(i) = 0) \wedge$
$\quad (pc(i) = S \wedge i = given \Rightarrow$
$\qquad request(i) = 0 \vee rbuffer(i) = 0)) \wedge$
$(waiting = i \Rightarrow$
$\quad given \neq i \wedge pc_i = S \wedge request_i = 0 \wedge rbuffer(i) = 0 \wedge$
$\quad sbuffer_i = 0 \wedge answer(i) = 0) \wedge$
$(sbuffer(i) = 1 \Rightarrow$
$\quad answer(i) = 0 \wedge request(i) = 0 \wedge rbuffer(i) = 0)$

As usual, the invariant has been elaborated in the course of the proof.

The RISC ProofNavigator Theory

```
newcontext "clientServer";

Index: TYPE = SUBTYPE(LAMBDA(x:INT): x=1 OR x=2);
Index0: TYPE = SUBTYPE(LAMBDA(x:INT): x=0 OR x=1 OR x=2);

% program counter type
PCBASE: TYPE;
R: PCBASE; S: PCBASE; C: PCBASE;
PC: TYPE = SUBTYPE(LAMBDA(x:PCBASE): x=R OR x=S OR x=C);
PCs: AXIOM R /= S AND R /= C AND S /= C;

% client states
pc: Index->PC; pc0: Index->PC;
request: Index->BOOLEAN; request0: Index->BOOLEAN;
answer: Index->BOOLEAN; answer0: Index->BOOLEAN;

% server state
given: Index0; given0: Index0;
waiting: Index0; waiting0: Index0;
sender: Index0; sender0: Index0;
rbuffer: Index -> BOOLEAN; rbuffer0: Index -> BOOLEAN;
sbuffer: Index -> BOOLEAN; sbuffer0: Index -> BOOLEAN;
```


The RISC ProofNavigator Theory (Contd)

```
% ----------------------------------------------------------------
% initial state condition
% ----------------------------------------------------------------

IC: (PC, BOOLEAN, BOOLEAN) -> BOOLEAN =
  LAMBDA(pc: PC, request: BOOLEAN, answer: BOOLEAN):
    pc = R AND (request <=> FALSE) AND (answer <=> FALSE);

IS: (Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN) -> BOOLEAN =
  LAMBDA(given: Index0, waiting: Index0, sender: Index0,
         rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN):
    given = 0 AND waiting = 0 AND sender = 0 AND
    (FORALL(i:Index): (rbuffer(i)<=>FALSE) AND (sbuffer(i)<=>FALSE));

Initial: BOOLEAN =
  (FORALL(i:Index): IC(pc(i), request(i), answer(i))) AND
  IS(given, waiting, sender, rbuffer, sbuffer);
```


The RISC ProofNavigator Theory (Contd'2)

```
% ----------------------------------------------------------------
% transition relation
% ----------------------------------------------------------------

RC: (PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN)->BOOLEAN =
  LAMBDA(pc: PC, request: BOOLEAN, answer: BOOLEAN,
         pc0: PC, request0: BOOLEAN, answer0: BOOLEAN):
    (pc = R AND (request <=> FALSE) AND
       pc0 = S AND (request0 <=> TRUE) AND (answer0 <=> answer)) OR
    (pc = S AND (answer <=> TRUE) AND
       pc0 = C AND (request0 <=> request) AND (answer0 <=> FALSE)) OR
    (pc = C AND (request <=> FALSE) AND
       pc0 = R AND (request0 <=> TRUE) AND (answer0 <=> answer));

RS: (Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN,
     Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN)->BOOLEAN =
  LAMBDA(given: Index0, waiting: Index0, sender: Index0,
         rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN,
         given0: Index0, waiting0: Index0, sender0: Index0,
         rbuffer0: Index->BOOLEAN, sbuffer0: Index->BOOLEAN):
```


The RISC ProofNavigator Theory (Contd'3)

```
   (EXISTS(i:Index):
       sender = 0 AND (rbuffer(i) <=> TRUE) AND
       sender0 = i AND (rbuffer0(i) <=> FALSE) AND
       given = given0 AND waiting = waiting0 AND sbuffer = sbuffer0 AND
       (FORALL(j:Index): j /= i => (rbuffer(j) <=> rbuffer0(j)))) OR
    (sender /= 0 AND sender = given AND waiting = 0 AND
       given0 = 0 AND sender0 = 0 AND
       waiting = waiting0 AND rbuffer = rbuffer0 AND sbuffer = sbuffer0) OR
    (sender /= 0 AND
       sender = given AND waiting /= 0 AND
       (sbuffer(waiting) <=> FALSE) AND
       given0 = waiting AND waiting0 = 0 AND
       (sbuffer0(waiting)<=>TRUE) AND (sender0 = 0) AND
       (rbuffer = rbuffer0) AND
       (FORALL(j:Index): j /= waiting => (sbuffer(j) <=> sbuffer0(j)))) OR
    (sender /= 0 AND (sbuffer(sender) <=> FALSE) AND
       sender /= given AND given = 0 AND given0 = sender AND
       (sbuffer0(sender)<=>TRUE) AND sender0=0 AND
       (waiting=waiting0) AND (rbuffer=rbuffer0) AND
       (FORALL(j:Index): j/= sender => (sbuffer(j) <=> sbuffer0(j)))) OR
    (sender /= 0 AND sender /= given AND given /= 0 AND
       waiting0 = sender AND sender0 = 0 AND
       given = given0 AND rbuffer = rbuffer0 AND sbuffer = sbuffer0);
```

The RISC ProofNavigator Theory (Contd'4)

```
External: (Index, PC, BOOLEAN, BOOLEAN, PC, BOOLEAN, BOOLEAN,
           Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN,
           Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN)->BOOLEAN =
  LAMBDA(i:Index,
         pc: PC, request: BOOLEAN, answer: BOOLEAN,
         pc0: PC, request0: BOOLEAN, answer0: BOOLEAN,
         given: Index0, waiting: Index0, sender: Index0,
           rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN,
         given0: Index0, waiting0: Index0, sender0: Index0,
           rbuffer0: Index->BOOLEAN, sbuffer0: Index->BOOLEAN):
    ((request <=> TRUE) AND
       pc0 = pc AND (request0 <=> FALSE) AND (answer0 <=> answer) AND
         (rbuffer0(i) <=> TRUE) AND given = given0 AND waiting = waiting0
         AND sender = sender0 AND sbuffer = sbuffer0 AND
         (FORALL (j: Index): j /= i => (rbuffer(j) <=> rbuffer0(j)))) OR
    (pc0 = pc AND (request0 <=> request) AND (answer0 <=> TRUE) AND
     (sbuffer(i) <=> TRUE) AND (sbuffer0(i) <=> FALSE) AND
     given = given0 AND waiting = waiting0 AND sender = sender0 AND
     rbuffer = rbuffer0 AND
     (FORALL (j: Index): j /= i => (sbuffer(j) <=> sbuffer0(j))));
```


The RISC ProofNavigator Theory (Contd'5)

```
Next: BOOLEAN =
  ((EXISTS (i: Index):
      RC(pc(i), request(i), answer(i),
         pc0(i), request0(i), answer0(i)) AND
     (FORALL (j: Index): j /= i =>
        pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
        (answer(j) <=> answer0(j)))) AND
    given = given0 AND waiting = waiting0 AND sender = sender0 AND
    rbuffer = rbuffer0 AND sbuffer = sbuffer0) OR
  (RS(given, waiting, sender, rbuffer, sbuffer,
      given0, waiting0, sender0, rbuffer0, sbuffer0) AND
   (FORALL (j:Index): pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
      (answer(j) <=> answer0(j)))) OR
  (EXISTS (i: Index):
    External(i, pc(i), request(i), answer(i),
                pc0(i), request0(i), answer0(i),
             given, waiting, sender, rbuffer, sbuffer,
             given0, waiting0, sender0, rbuffer0, sbuffer0) AND
   (FORALL (j: Index): j /= i =>
      pc(j) = pc0(j) AND (request(j) <=> request0(j)) AND
      (answer(j) <=> answer0(j))));
```


The RISC ProofNavigator Theory (Contd'6)

```
% ----------------------------------------------------------------------------
% invariant
% ----------------------------------------------------------------------------
Invariant: (Index->PC, Index->BOOLEAN, Index->BOOLEAN,
            Index0, Index0, Index0, Index->BOOLEAN, Index->BOOLEAN) -> BOOLEAN =
  LAMBDA(pc: Index->PC, request: Index->BOOLEAN, answer: Index->BOOLEAN,
         given: Index0, waiting: Index0, sender: Index0,
         rbuffer: Index->BOOLEAN, sbuffer: Index->BOOLEAN):
    FORALL (i: Index):
      (pc(i) = C OR (sbuffer(i) <=> TRUE) OR (answer(i) <=> TRUE) =>
         given = i AND
         (FORALL (j: Index): j /= i =>
            pc(j) /= C AND
            (sbuffer(j) <=> FALSE) AND (answer(j) <=> FALSE))) AND
      (pc(i) = R =>
         (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE) AND
         (i /= given =>
           (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i)
           AND
         (i = given =>
           (request(i) <=> TRUE) OR (rbuffer(i) <=> TRUE) OR sender = i) AND
         ((request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND
```


The RISC ProofNavigator Theory (Contd'7)

```
      (pc(i) = S =>
         ((sbuffer(i) <=> TRUE) OR (answer(i) <=> TRUE) =>
            (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i)
            AND
         (i /= given =>
            (request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND
      (pc(i) = C =>
        (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE) AND sender /= i AND
        (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE)) AND
      (sender = 0 AND ((request(i) <=> TRUE) OR (rbuffer(i) <=> TRUE)) =>
        (sbuffer(i) <=> FALSE) AND (answer(i) <=> FALSE)) AND
      (sender = i =>
        (sender = given AND pc(i) = R =>
           (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE)) AND
        waiting /= i AND
        (pc(i) = S AND i /= given =>
           (request(i) <=> FALSE) AND (rbuffer(i) <=> FALSE)) AND
        (pc(i) = S AND i = given =>
           (request(i) <=> FALSE) OR (rbuffer(i) <=> FALSE))) AND
```

The RISC ProofNavigator Theory (Contd'8)

```
(waiting = i =>
   given /= i AND
   pc(waiting) = S AND
   (request(waiting) <=> FALSE) AND (rbuffer(waiting) <=> FALSE) AND
   (sbuffer(waiting) <=> FALSE) AND (answer(waiting) <=> FALSE)) AND
((sbuffer(i) <=> TRUE) =>
   (answer(i) <=> FALSE) AND (request(i) <=> FALSE) AND
     (rbuffer(i) <=> FALSE));
```


The RISC ProofNavigator Theory (Contd'9)

```
% ----------------------------------------------------------------------
% mutual exclusion proof
% ----------------------------------------------------------------------
MutEx: FORMULA
  Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer) =>
  NOT(pc(1) = C AND pc(2) = C);

% ----------------------------------------------------------------------
% invariance proof
% ----------------------------------------------------------------------
Inv1: FORMULA
  Initial =>
    Invariant(pc, request, answer, given, waiting, sender, rbuffer, sbuffer);

Inv2: FORMULA
  Invariant(pc, request, answer, given, waiting, sender,
    rbuffer, sbuffer) AND Next =>
  Invariant(pc0, request0, answer0, given0, waiting0, sender0,
    rbuffer0, sbuffer0);
```


The Proofs: MutEx and Inv1

```
[z3f]: expand Invariant, IC, IS
  [nhn]: scatter
    [znj]: auto
      [n1u]: proved (CVCL)
```

Single application of `autostar`.

```
[oas]: expand Initial, Invariant, IC, IS
  [eij]: scatter
    [5ul]: auto
      [uvj]: proved (CVCL)
    [6ul]: auto
      [2u6]: proved (CVCL)
    [avl]: auto
      [cuv]: proved (CVCL)
    [bvl]: auto
      [jtl]: proved (CVCL)
    [cvl]: auto
      [qsb]: proved (CVCL)
    [dvl]: auto
      [xrx]: proved (CVCL)
    [evl]: auto
      [5qn]: proved (CVCL)
    [fvl]: auto
      [fqd]: proved (CVCL)
    [gvl]: auto
      [mpz]: proved (CVCL)
    [hvl]: proved (CVCL)
    [h5h]: auto
      [p3z]: proved (CVCL)
    [i5h]: auto
      [gjb]: proved (CVCL)
    [j5h]: auto
      [4vi]: proved (CVCL)
    [k5h]: auto
      [ucq]: proved (CVCL)
    [l5h]: auto
      [lpx]: proved (CVCL)
    [m5h]: proved (CVCL)
    [n5h]: proved (CVCL)
    [o5h]: proved (CVCL)
    [p5h]: proved (CVCL)
    [q5h]: proved (CVCL)
    [q5i]: proved (CVCL)
    [r5i]: proved (CVCL)
    [s5i]: proved (CVCL)
    [t5i]: proved (CVCL)
    [u5i]: auto
      [1br]: proved (CVCL)
    [v5i]: auto
      [roy]: proved (CVCL)
    [w5i]: auto
      [i26]: proved (CVCL)
    [x5i]: proved (CVCL)
    [y5i]: auto
      [wuo]: proved (CVCL)
    [z5i]: auto
      [nbw]: proved (CVCL)
    [z5j]: auto
      [nbn]: proved (CVCL)
    [15j]: auto
      [eou]: proved (CVCL)
    [25j]: proved (CVCL)
      [35j]: proved (CVCL)
    [45j]: proved (CVCL)
    [55j]: proved (CVCL)
    [65j]: proved (CVCL)
```


The Proofs: Inv2

```
[pas]: scatter
  [lbh]: expand Next
    [pzi]: split bfv
      [leh]: decompose
        [pkr]: expand RS
          [lpn]: split 5xv
            [pt6]: expand Invariant
              [lcw]: scatter
                [puh]: auto
                  [143]: proved (CVCL)
                ... (20 times)
                [tuh]: proved (CVCL)
                ... (15 times)
            [qt6]: expand Invariant
              [snq]: scatter
                [avi]: auto
                  [cct]: proved (CVCL)
                ... (26 times)
                [gvi]: proved (CVCL)
                ... (6 times)
            [rt6]: scatter
              [zyk]: expand Invariant
                [rvj]: scatter
                  [zgj]: auto
                    [rhd]: proved (CVCL)
                  ... (31 times)
                  [2f3]: proved (CVCL)
                  ... (1 times)
```

```
[st6]: scatter
  [aef]: expand Invariant
    [cwk]: scatter
      [q16]: auto
        [seg]: proved (CVCL)
      ... (21 times)
      [w16]: proved (CVCL)
      ... (12 times)
[tt6]: scatter
  [hp6]: expand Invariant
    [twl]: scatter
      [hqv]: auto
        [tbj]: proved (CVCL)
      ... (27 times)
      [nqv]: proved (CVCL)
      ... (6 times)
[meh]: scatter
  [w3z]: expand External
    [3rk]: split lhe
      [g4b]: scatter
        [mdh]: expand Invariant
          [wzf]: scatter
            [3ys]: auto
              [gsh]: proved (CVCL)
            ... (36 times)
```

```
[h4b]: scatter
  [tob]: expand Invariant
    [h1g]: scatter
      [t4i]: auto
        [hpk]: proved (CVCL)
      ... (36 times)
[neh]: scatter
  [4oc]: expand RC
    [nuh]: split nwz
      [4ge]: scatter
        [ney]: expand Invariant
          [45d]: scatter
            [nui]: auto
              [4wr]: proved (CVCL)
            ... (36 times)
          [5ge]: scatter
            [ups]: expand Invariant
              [o6e]: scatter
                [ez5]: auto
                  [5tu]: proved (CVCL)
                ... (36 times)
          [6ge]: scatter
            [21m]: expand Invariant
              [66f]: scatter
                [24u]: auto
                  [6qx]: proved (CVCL)
                ... (36 times)
```

Ten main branches each requiring only single application of `autostar`.

1. Verification by Computer-Supported Proving

2. The Model Checker Spin

3. Verification by Automatic Model Checking

The Model Checker Spin

- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Workshop series since 1995 (12th workshop "Spin 2005").
  - ACM System Software Award in 2001.
- Spin resources:
  - Web site: http://spinroot.com.
  - Survey paper: Holzmann "The Model Checker Spin", 1997.
  - Book: Holzmann "The Spin Model Checker — Primer and Reference Manual", 2004.

Goal: verification of (concurrent/distributed) software models.

The Model Checker Spin

On-the-fly LTL model checking of finite state systems.

- System S modeled by automaton S_A.
  - Explicit representation of automaton states.
  - There exist various other approaches (discussed later).
- On-the-fly model checking.
  - Reachable states of S_A are only expended on demand.
  - *Partial order reduction* to keep state space manageable.
- LTL model checking.
  - Property P to be checked described in PLTL.
    - Propositional linear temporal logic.
  - Description converted into property automaton P_A.
    - Automaton accepts only system runs that do not satisfy the property.

Model checking based on automata theory.

The Spin System Architecture

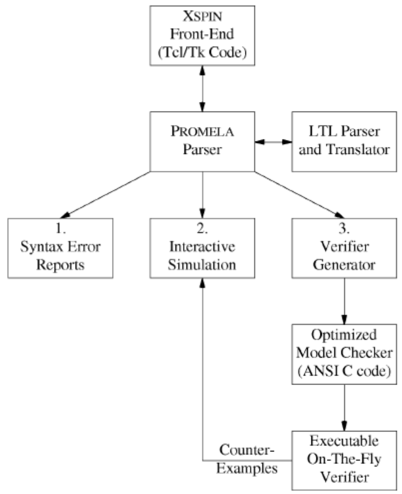


Fig. 1. The structure of SPIN simulation and verification.

Features of Spin

- System description in Promela.
  - Promela = Process Meta-Language.
    - Spin = Simple Promela Interpreter.
  - Express coordination and synchronization aspects of a real system.
  - Actual computation can be e.g. handled by embedded C code.
- Simulation mode.
  - Investigate individual system behaviors.
  - Inspect system state.
  - Graphical interface XSpin for visualization.
- Verification mode.
  - Verify properties shared by all possible system behaviors.
  - Properties specified in PLTL and translated to "never claims".
    - Promela description of automaton for negation of the property.
  - Generated counter examples may be investigated in simulation mode.

Verification and simulation are tightly integrated in Spin.

The Client/Server System in Promela

```
/* definition of a constant MESSAGE */
mtype = { MESSAGE };

/* two arrays of channels of size 2,
   each channel has a buffer size 1 */
chan request[2] = [1] of { mtype };
chan answer  [2] = [1] of { mtype };

/* two global arrays for monitoring
   the states of the clients */
bool inC[2]  = false;
bool wait[2] = false;

/* the system of three processes */
init
{
  run client(1);
  run client(2);
  run server();
}
```

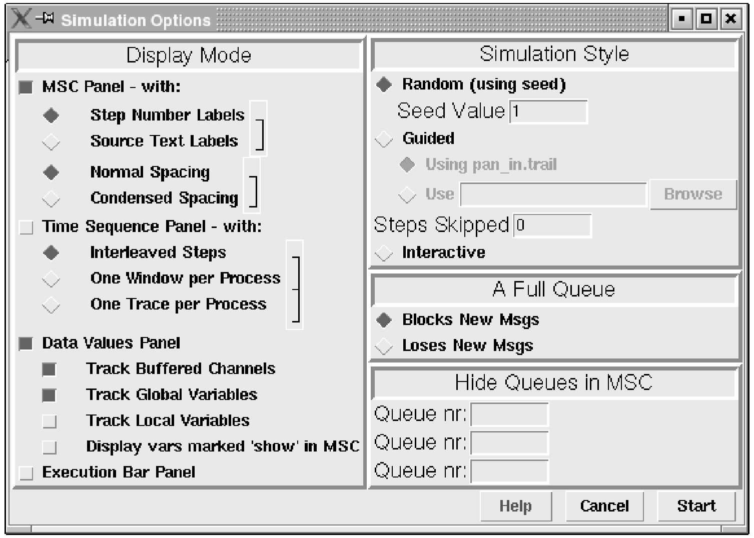
```
/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;

    wait[id-1] = true;
    answer[id-1]  ? MESSAGE;
    wait[id-1] = false;

    inC[id-1] = true;
    skip; // the critical region
    inC[id-1] = false;

    request[id-1] ! MESSAGE
  od;
}
```


The Client/Server System in Promela

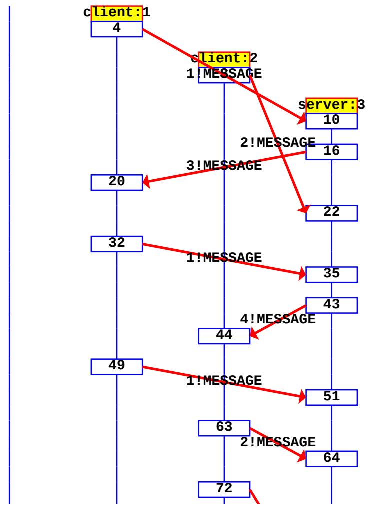
```
/* the server process type */
proctype server()
{
  /* three variables of two bit each */
  unsigned given   : 2 = 0;
  unsigned waiting : 2 = 0;
  unsigned sender  : 2;

  do :: true ->

    /* receiving the message */
    if
    :: request[0] ? MESSAGE ->
      sender = 1
    :: request[1] ? MESSAGE ->
      sender = 2
    fi;
```

```
    /* answering the message */
    if
    :: sender == given ->
      if
      :: waiting == 0 ->
        given = 0
      :: else ->
        given = waiting;
        waiting = 0;
        answer[given-1] ! MESSAGE
      fi;
    :: given == 0 ->
      given = sender;
      answer[given-1] ! MESSAGE
    :: else
      waiting = sender
    fi;

  od;
}
```

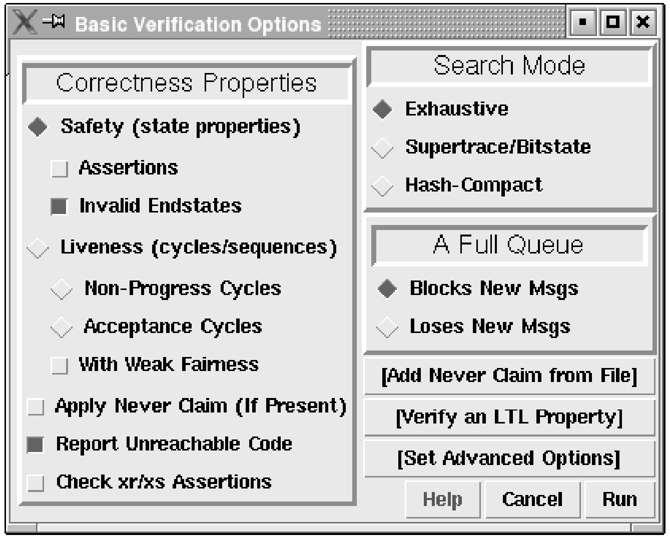

Spin Simulation Options



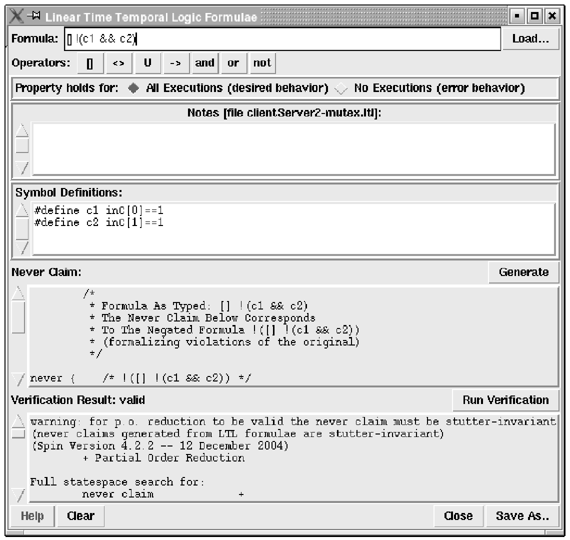
Simulating the System Execution in Spin



Spin Verification Options



Specifying a System Property in Spin



Spin Verification Output

```
(Spin Version 4.2.2 -- 12 December 2004)
+ Partial Order Reduction

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

State-vector 48 byte, depth reached 477, errors: 0
     499 states, stored
     395 states, matched
     894 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
...
0.00user 0.01system 0:00.01elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+737minor)pagefaults 0swaps
```

More Promela Features

Active processes, inline definitions, atomic statements, output.

```
mtype = { P, C, N }
mtype turn = P;

inline request(x, y) { atomic { x == y -> x = N } }
inline release(x, y) { atomic { x = y } }
#define FORMAT "Output: %s\n"

active proctype producer()
{
  do
  :: request(turn, P) -> printf(FORMAT, "P"); release(turn, C);
  od
}

active proctype consumer()
{
  do
  :: request(turn, C) -> printf(FORMAT, "C"); release(turn, P);
  od
}
```


More Promela Features

Embedded C code.

```
/* declaration is added locally to proctype main */
c_state "float f" "Local main"

active proctype main()
{
  c_code { Pmain->f = 0; }
  do
    :: c_expr { Pmain->f <= 300 };
       c_code { Pmain->f = 1.5 * Pmain->f ; };
       c_code { printf("%4.0f\n", Pmain->f); };
  od;
 }
```

Can embed computational aspects into a Promela model (only works in verification mode where a C program is generated from the model).

Command-Line Usage for Simulation

Command-line usage of spin: `spin --`.

- Perform syntax check.
  `spin -a` *`file`*
- Run simulation.

No output:	`spin` *`file`*
One line per step:	`spin -p` *`file`*
One line per message:	`spin -c` *`file`*
Bounded simulation:	`spin -u`*`steps file`*
Reproducible simulation:	`spin -n`*`seed file`*
Interactive simulation:	`spin -i` *`file`*
Guided simulation:	`spin -t` *`file`*

Command-Line Usage for Verification

- Generate never claim
  `spin -f "`*`nformula`*`" >`*`neverfile`*
- Generate verifier.
  `spin -N` *`neverfile`* `-a` *`file`*

```
ls -la pan.*
-rw-r--r--  1 schreine schreine   3073 2005-05-10 16:36 pan.b
-rw-r--r--  1 schreine schreine 150665 2005-05-10 16:36 pan.c
-rw-r--r--  1 schreine schreine   8735 2005-05-10 16:36 pan.h
-rw-r--r--  1 schreine schreine  14163 2005-05-10 16:36 pan.m
-rw-r--r--  1 schreine schreine  19376 2005-05-10 16:36 pan.t
```

- Compile verifier.
  `cc -O3 -DMEMLIM=128 -o pan pan.c`
- Execute verifier.

Options:	`./pan --`
Find acceptance cycle:	`./pan -a`
Weak scheduling fairness:	`./pan -a -f`
Maximum search depth:	`./pan -a -f -m`*`depth`*

Spin Verifier Generation Options

`cc -O3 options -o pan pan.c`

-DNP	Include code for non-progress cycle detection
-DMEMLIM=*N*	Maximum number of MB used
-DNOREDUCE	Disable partial order reduction
-DCOLLAPSE	Use collapse compression method
-DHC	Use hash-compact method
-DDBITSTATE	Use bitstate hashing method

For detailed information, look up the manual.

The Basic Approach

Translation of the original problem to a problem in automata theory.

- Original problem: $S \models P$.
  - $S = \langle I, R \rangle$, PLTL formula P.
  - Does property P hold for every run of system S?
- Construct system automaton S_A with language $\mathcal{L}(S_A)$.
  - A language is a set of infinite words.
  - Each such word describes a system run.
  - $\mathcal{L}(S_A)$ describes the set of runs of S.
- Construct property automaton P_A with language $\mathcal{L}(P_A)$.
  - $\mathcal{L}(P_A)$ describes the set of runs satisfying P.
- Equivalent Problem: $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.
  - The language of S_A must be contained in the language of P_A.

There exists an efficient algorithm to solve this problem.

Finite State Automata

A (variant of a) labeled transition system in a finite state space.

- Take finite sets *State* and *Label*.
  - The state space *State*.
  - The alphabet *Label*.
- A (finite state) automaton $A = \langle I, R, F \rangle$ over *State* and *Label*:
  - A set of initial states $I \subseteq State$.
  - A labeled transition relation $R \subseteq Label \times State \times State$.
  - A set of final states $F \subseteq State$.
    - Büchi automata: F is called the set of accepting states.

We will only consider infinite runs of Büchi automata.

Runs and Languages

- An infinite run $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ of automaton A:
  - $s_0 \in I$ and $R(l_i, s_i, s_{i+1})$ for all $i \in \mathbb{N}$.
  - Run r is said to read the infinite word $w(r) := \langle l_0, l_1, l_2, \ldots \rangle$.
- $A = \langle I, R, F \rangle$ accepts an infinite run r:
  - Some state $s \in F$ occurs infinitely often in r.
  - This notion of acceptance is also called Büchi acceptance.
- The language $\mathcal{L}(A)$ of automaton A:
  - $\mathcal{L}(A) := \{w(r) : A \text{ accepts } r\}$.
  - The set of words which are read by the runs accepted by A.
- Example: $\mathcal{L}(A) = (a^*bb^*a)^*a^\omega + (a^*bb^*a)^\omega = (b^*a)^\omega$.
  - $w^i = ww\ldots w$ (i occurrences of w).
  - $w^* = \{w^i : i \in \mathbb{N}\} = \{\langle\rangle, w, ww, www, \ldots\}$.
  - $w^\omega = wwww\ldots$ (infinitely often).
  - An infinite repetition of an arbitrary number of b followed by a.

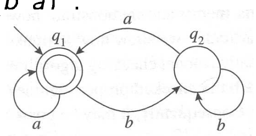


Figure 9.1
A finite automaton.

Edmund Clarke: "Model Checking", 1999.

A Finite State System as an Automaton

The automaton $S_A = \langle I, R, F \rangle$ for a finite state system $S = \langle I_S, R_S \rangle$:
- $State := States_S \cup \{\iota\}$.
  - The state space $States_S$ of S is finite; additional state ι ("iota").
- $Label := \mathbb{P}(AP)$.
  - Finite set AP of atomic propositions.

    All PLTL formulas are built from this set only.
  - Powerset $\mathbb{P}(S) := \{s : s \subseteq S\}$.
  - Every element of $Label$ is thus a set of atomic propositions.
- $I := \{\iota\}$.
  - Single initial state ι.
- $R(l, s, s') :\Leftrightarrow l = L(s') \wedge (R_S(s, s') \vee (s = \iota \wedge I_S(s')))$.
  - $L(s) := \{p \in AP : s \models p\}$.
  - Each transition is labeled by the set of atomic propositions satisfied by the successor state.
  - Thus all atomic propositions are evaluated on the successor state.
- $F := State$.
  - Every state is accepting.

A Finite State System as an Automaton

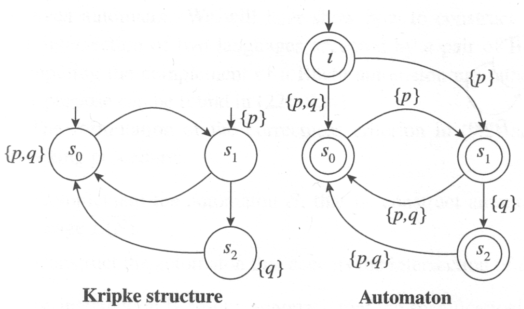


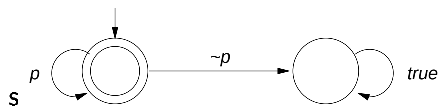
Figure 9.2
Transforming a Kripke structure into an automaton.

Edmund Clarke et al: "Model Checking", 1999.

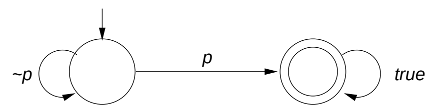
If $r = s_0 \to s_1 \to s_2 \to \ldots$ is a run of S, then S_A accepts the labelled version $r_l := \iota \xrightarrow{L(s_0)} s_0 \xrightarrow{L(s_1)} s_1 \xrightarrow{L(s_2)} s_2 \xrightarrow{L(s_3)} \ldots$ of r.

A System Property as an Automaton

Also an PLTL formula can be translated to a finite state automaton.
- We need the automaton P_A for a PLTL property P.
  - Requirement: $r \models P \Leftrightarrow P_A$ accepts r_l.
  - A run satisfies property P if and only if automaton A_P accepts the labeled version of the run.
- Example: $\Box p$.

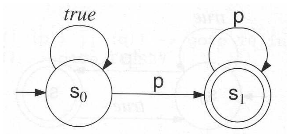


- Example: $\Diamond p$.



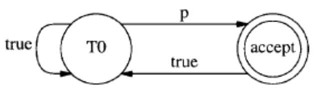
Further Examples

- Example: $\Diamond\Box p$.



Gerard Holzmann: "The Spin Model Checker", 2004.

- Example: $\Box\Diamond p$.



Gerard Holzmann: "The Model Checker Spin", 1997.

Arbitrary PLTL formulas can be converted to automata.

System Properties

- State equivalence: $L(s) = L(t)$.
  - Both states have the same labels.
  - Both states satisfy the same atomic propositions in AP.
- Run equivalence: $w(r_l) = w(r'_l)$.
  - Both runs have the same sequences of labels.
  - Both runs satisfy the same PLTL formulas built over AP.
- Indistinguishability: $w(r_l) = w(r'_l) \Rightarrow (r \models P \Leftrightarrow r' \models P)$
  - PLTL formula P cannot distinguish between runs r and r' whose labeled versions read the same words.
- Consequence: $S \models P \Leftrightarrow \mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.
  - Proof that, if every run of S satisfies P, then every word $w(r_l)$ in $\mathcal{L}(S_A)$ equals some word $w(r'_l)$ in $\mathcal{L}(P_A)$, and vice versa.
  - "Vice versa" direction relies on indistinguishability property.

The Next Steps

- Problem: $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$
  - Equivalent to: $\mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$.
    - Complement $\overline{L} := \{w : w \notin L\}$.
  - Equivalent to: $\mathcal{L}(S_A) \cap \mathcal{L}(\neg P_A) = \emptyset$.
    - $\overline{\mathcal{L}(A)} = \mathcal{L}(\neg A)$.
- Equivalent Problem: $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$.
  - We will introduce the synchronized product automaton $A \otimes B$.
    - A transition of $A \otimes B$ represents a simultaneous transition of A and B.
  - Property: $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$.
- Final Problem: $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$.
  - We have to check whether the language of this automaton is empty.
  - We have to look for a word w accepted by this automaton.
    - If no such w exists, then $S \models P$.
    - If such a $w = w(r_l)$ exists, then r is a counterexample, i.e. a run of S such that $r \not\models P$.

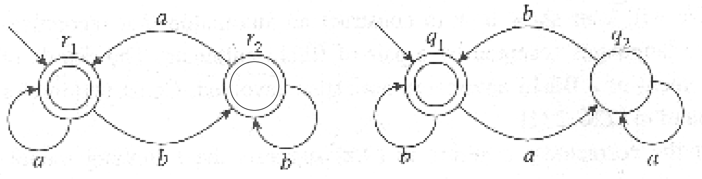
Synchronized Product of Two Automata

Given two finite automata $A = \langle I_A, R_A, State_A \rangle$ and $B = \langle I_B, R_B, F_B \rangle$.

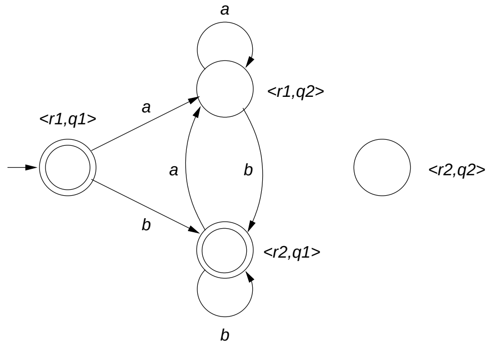
- Synchronized product $A \otimes B = \langle I, R, F \rangle$.
  - $State := State_A \times State_B$.
  - $Label := Label_A = Label_B$.
  - $I := I_A \times I_B$.
  - $R(l, \langle s_A, s_B \rangle, \langle s'_A, s'_B \rangle) :\Leftrightarrow R_A(l, s_A, s'_A) \land R_B(l, s_B, s'_B)$.
  - $F := State_A \times F_B$.

Special case where all states of automaton A are accepting.

Synchronized Product of Two Automata

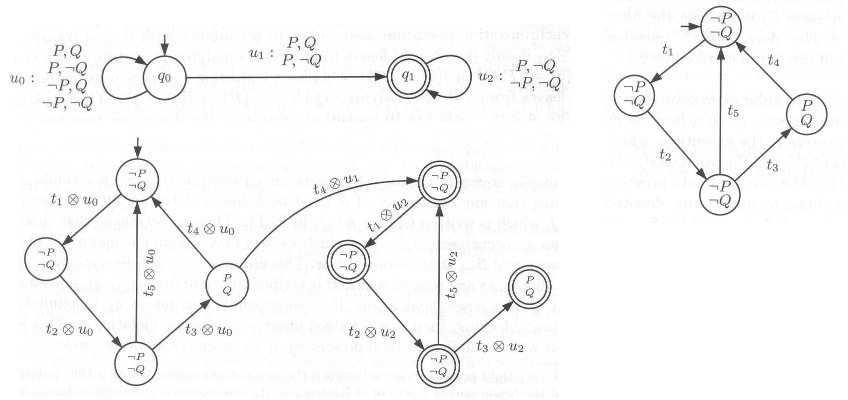


Edmund Clarke: "Model Checking", 1999.



Example

Check whether $S \models \Box(P \Rightarrow \bigcirc\Diamond Q)$.



B. Berard et al: "Systems and Software Verification", 2001.

The product automaton accepts a run, thus the property does not hold.

Checking Emptiness

How to check whether $\mathcal{L}(A)$ is non-empty?

- Suppose $A = \langle I, R, F \rangle$ accepts a run r.
  - Then r contains infinitely many occurrences of some state in F.
  - Since *State* is finite, in some suffix r' every state occurs infinit. often.
  - Thus every state in r' is reachable from every other state in r'.
- C is a strongly connected component (SCC) of graph G if
  - C is a subgraph of G,
  - every node in C is reachable from every other node in C along a path entirely contained in C, and
  - C is maximal (not a subgraph of any other SCC of G).
- Thus the states in r' are contained in an SCC C.
  - C is reachable from an initial state.
  - C contains an accepting state.
  - Conversely, *any* such SCC generates an accepting run.

$\mathcal{L}(A)$ is non-empty if and only if the reachability graph of A has an SCC that contains an accepting state.

Checking Emptiness

Find in the reachability graph an SCC that contains an accepting state.

- We have to find an accepting state with a cycle back to itself.
  - Any such state belongs to some SCC.
  - Any SCC with an accepting state has such a cycle.
  - Thus this is a sufficient and necessary condition.
- Any such a state s defines a counterexample run r.
  - $r = \iota \to \ldots \to s \to \ldots \to s \to \ldots \to s \to \ldots$
  - Finite prefix $\iota \to \ldots \to s$ from initial state ι to s.
  - Infinite repetition of cycle $s \to \ldots \to s$ from s to itself.

This is the core problem of PLTL model checking; it can be solved by a *depth-first search* algorithm.

Basic Structure of Depth-First Search

Visit all states of the reachability graph of an automaton $\langle \{\iota\}, R, F \rangle$.

```
global
  StateSpace V := {}
  Stack D := ⟨⟩

proc main()
  push(D, ι)
  visit(ι)
  pop(D)
end
```

```
proc visit(s)
  V := V ∪ {s}
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      visit(s')
      pop(D)
    end
  end
end
```

State space V holds all states visited so far; stack D holds path from initial state to currently visited state.

Checking State Properties

Apply depth-first search to checking a state property (assertion).

```
global
  StateSpace V := {}
  Stack D := ⟨⟩

proc main()
  // r becomes true, iff
  // counterexample run is found
  push(D, ι)
  r := search(ι)
  pop(D)
end
```

```
function search(s)
  V := V ∪ {s}
  if ¬check(s) then
    print D
    return true
  end
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      r := search(s')
      pop(D)
      if r then return true end
    end
  end
  return false
end
```

Stack D can be used to print counterexample run.

Depth-First Search for Acceptance Cycle

```
global
  ...
  Stack C := ⟨⟩

proc main()
  push(D, ι); r := search(ι); pop(D)
end

function searchCycle(s)
  for ⟨l, s, s'⟩ ∈ R do
    if has(D, s') then
      print D; print C; print s'
      return true
    else if ¬has(C, s') then
      push(C, s');
      r := searchCycle(s')
      pop(C);
      if r then return true end
    end
  end
  return false
end
```

```
boolean search(s)
  V := V ∪ {s}
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      r := search(s')
      pop(D)
      if r then return true end
    end
  end
  if s ∈ F then
    r := searchCycle(s)
    if r then return true end
  end
  return false
end
```


Depth-First Search for Acceptance Cycle

- At each call of *search*(s),
  - s is a reachable state,
  - D describes a path from ι to s.
- *search* calls *searchCycle*(s)
  - on a reachable accepting state s
  - in order to find a cycle from s to itself.
- At each call of *searchCycle*(s),
  - s is a state reachable from a reachable accepting state s_a,
  - D describes a path from ι to s_a,
  - $D \rightarrow C$ describes a path from ι to s (via s_a).
- Thus we have found an accepting cycle $D \rightarrow C \rightarrow s'$, if
  - there is a transition $s \xrightarrow{l} s'$,
  - such that s' is contained in D.

If the algorithm returns "true", there exists a violating run; the converse follows from the exhaustiveness of the search.

Implementing the Search

- The state space V,
  - is implemented by a hash table for efficiently checking $s' \notin V$.
- Rather than using explicit stacks D and C,
  - each state node has two bits d and c,
  - d is set to denote that the state is in stack D,
  - c is set to denote that the state is in stack C.
- The counterexample is printed,
  - by searching, starting with ι, the unique sequence of reachable nodes where d is set until the accepting node s_a is found, and
  - by searching, starting with a successor of s_a, the unique sequence of reachable nodes where c is set until the cycle is detected.
- Furthermore, it is not necessary to reset the c bits, because
  - *search* first explores all states reachable by an accepting state s before trying to find a cycle from s; from this, one can show that
  - called with the first accepting node s that is reachable from itself, *search2* will not encounter nodes with c bits set in previous searches.
  - With this improvement, every state is only visited twice.

Complexity of the Search

The complexity of checking $S \models P$ is as follows.

- Let $|P|$ denote the number of subformulas of P.
- $|State_{(\neg P)_A}| = O(2^{|P|})$.
- $|State_{A \otimes B}| = |State_A| \cdot |State_B|$.
- $|State_{S_A \otimes (\neg P)_A}| = O(|State_{S_A}| \cdot 2^{|P|})$
- The time complexity of *search* is linear in the size of *State*.
  - Actually, in the number of reachable states (typically much smaller).
  - Only true for the improved variant where the c bits are not reset.
  - Then every state is visited at most twice.

PLTL model checking is linear in the number of reachable states but exponential in the size of the formula.

The Overall Process

Basic PLTL model checking for deciding $S \models P$.

- Convert system S to automaton S_A.
  - Atomic propositions of PLTL formula are evaluated on each state.
- Convert negation of PLTL formula P to automaton $(\neg P)_A$.
  - How to do so, remains to be described.
- Construct synchronized product automaton $S_A \otimes (\neg P)_A$.
  - After that, formula labels are not needed any more.
- Find SCC in reachability-graph of product automaton.
  - A purely graph-theoretical problem that can be efficiently solved.
  - Time complexity is linear in the size of the state space of the system but exponential in the size of the formula to be checked.
  - Weak scheduling fairness with k components: runtime is increased by factor $k + 2$ (worst-case, "in practice just factor 2" [Holzmann]).

The basic approach immediately leads to *state space explosion*; further improvements are needed to make it practical.

On the Fly Model Checking

For checking $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$, it is not necessary to construct the states of S_A in advance.

- Only the property automaton $(\neg P)_A$ is constructed in advance.
  - This automaton has comparatively small state space.
- The system automaton S_A is constructed on the fly.
  - Construction is guided by $(\neg P)_A$ while computing $S_A \otimes (\neg P)_A$.
  - Only that part of the reachability graph of S_A is expanded that is consistent with $(\neg P)_A$ (i.e. can lead to a counterexample run).
- Typically only a part of the state space of S_A is investigated.
  - A smaller part, if a counterexample run is detected early.
  - A larger part, if no counterexample run is detected.

Unreachable system states and system states that are not along possible counterexample runs are never constructed.

On the Fly Model Checking

Expansion of state $s = \langle s_0, s_1 \rangle$ of product automaton $S_A \otimes (\neg P)_A$ into the set $R(s)$ of transitions from s (**for** $\langle l, s, s' \rangle \in R(s)$ **do** ...).

- Let S_1' be the set of all successors of state s_1 of $(\neg P)_A$.
  - Property automaton $(\neg P)_A$ has been precomputed.
- Let S_0' be the set of all successors of state s_0 of S_A.
  - Computed on the fly by applying system transition relation to s_0.
- $R(s) := \{\langle l, \langle s_0, s_1 \rangle, \langle s_0', s_1' \rangle \rangle : s_0' \in S_0' \wedge s_1' \in S_1' \wedge s_1 \xrightarrow{l} s_1' \wedge L(s_0') \in l\}$.
  - Choose candidate $s_0' \in S_0'$.
  - Determine set of atomic propositions $L(s_0')$ true in s_0'.
  - If $L(s_0')$ is not consistent with the label of any transition $\langle s_0, s_1 \rangle \xrightarrow{l} \langle s_0', s_1' \rangle$ of the proposition automaton, s_0' it is ignored.
  - Otherwise, R is extended by every transition $\langle s_0, s_1 \rangle \xrightarrow{l} \langle s_0', s_1' \rangle$ where $L(s_0')$ is consistent with label l of transition $s_1 \xrightarrow{l} s_1'$.

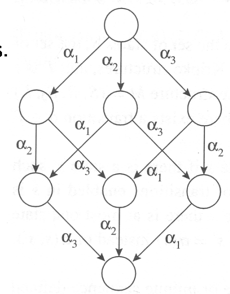
Actually, depth-first search proceeds with first suitable successor $\langle s_0', s_1' \rangle$ before expanding the other candidates.

Partial Order Reduction

Core problem of model checking: state space explosion.

- Take asynchronous composition $S_0 || S_1 || \ldots || S_{k-1}$.
  - Take state s where one transition of each component is enabled.
    - Assume that the transition of one component does not disable the transitions of the other components and that no other transition becomes enabled before all three transitions have been performed.
  - Take state s' after execution of all three transitions.
    - There are $k!$ paths leading from s to s'.
    - There are 2^k states involved in the transitions.

Sometimes it suffices to consider a *single path* with $k+1$ states.



Edmund Clarke: "Model Checking", 1999.

Partial Order Reduction

Check $S \models P$.

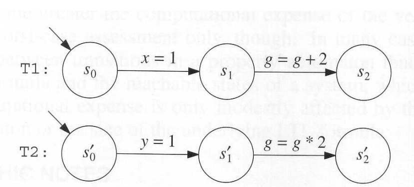
boolean *search*(s) ... **for** $\langle l, s, s' \rangle \in R(s)$ **do**	$\rightsquigarrow$	**boolean** *search*(s) ... **for** $\langle l, s, s' \rangle \in ample_P(s)$ **do**

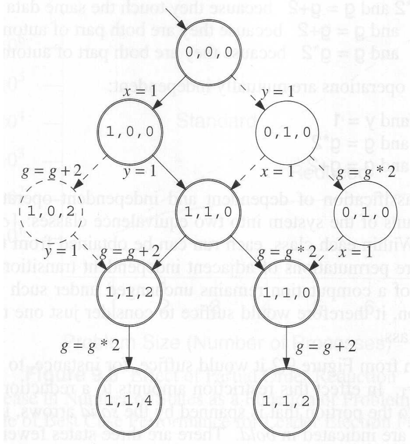
- $ample_P(s) \subseteq R(s)$.
  - The ample set $ample_P(s)$.
    - The set of transitions from s to be considered for checking P.
  - $R(s) := \{\langle l, s, s' \rangle : l \in Label \wedge s' \in State\}$.
    - The set of all transitions from s.
  - Optimization: $ample_P(s) \subsetneq R(s)$.
    - Search space is reduced.

There exists an algorithm for the calculation of the ample set.

Example

Check $(T1 || T2) \models \Diamond g \geq 2$.





Gerard Holzmann: "The Spin Model Checker", 1999.

For checking $\Diamond g \geq 2$, it suffices to check only one ordering of the independent transitions $x = 1$ and $y = 1$ (not true for checking $\Box x \geq y$).

Example

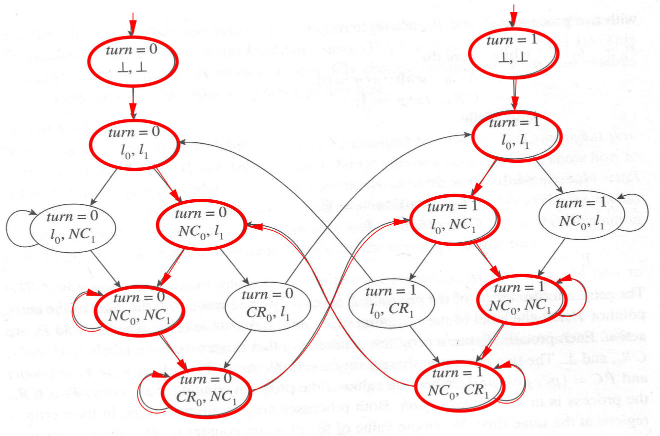


Figure 2.2
Reachable states of Kripke structure for mutual exclusion example.

Edmund Clarke et al: "Model Checking", 1999.

System after partial order reduction.

Other Optimizations

- Statement merging.
  - Special case of partial order reduction where a sequence of transitions of same component is combined to a single transition.
- State compression.
  - Collapse compression: each state holds pointers to component states; thus component states can be shared among many system states.
  - Minimized automaton representation: represent state set V not by hash table but by finite state automaton that accepts a state (sequence of bits) s if and only if $s \in V$.
  - Hash compact: store in the hash table a hash value of the state (computed by a different hash function). Probabilistic approach: fails if two states are mapped to the same hash value.
  - Bitstate hashing: represent V by a bit table whose size is much larger than the expected number of states; each state is then only represented by a single bit. Probabilistic approach: fails if two states are hashed to the same position in the table.

Other Approaches to Model Checking

There are fundamentally different approaches to model checking than the automata-based one implemented in Spin.

- Symbolic Model Checking (e.g. SMV, NuSMV).
  - Core: binary decision diagrams (BDDs).
    - Data structures to represent boolean functions.
    - Can be used to describe state sets and transition relations.
  - The set of states satisfying a CTL formula P is computed as the BDD representation of a fixpoint of a function (predicate transformer) F_P.
    - If all initial system states are in this set, P is a system property.
  - BDD packages for efficiently performing the required operations.
- Bounded Model Checking (e.g. NuSMV2).
  - Core: propositional satisfiability.
    - Is there a truth assignment that makes propositional formula true?
  - There is a counterexample of length at most k to a LTL formula P, if and only if a particular propositional formula $F_{k,P}$ is satisfiable.
    - Problem: find suitable bound k that makes method complete.
  - SAT solvers for efficiently deciding propositional satisfiability.

Other Approaches to Model Checking

- Counter-Example Guided Abstraction Refinement (e.g. BLAST).
  - Core: model abstraction.
    - A finite set of predicates is chosen and an abstract model of the system is constructed as a finite automaton whose states represent truth assignments of the chosen predicates.
  - The abstract model is checked for the desired property.
    - If the abstract model is error-free, the system is correct; otherwise an abstract counterexample is produced.
    - It is checked whether the abstract counterexample corresponds to a real counterexample; if yes, the system is not correct.
    - If not, the chosen set of predicates contains too little information to verify or falsify the program; new predicates are added to the set. Then the process is repeated.
  - Core problem: how to refine the abstraction.
    - Automated theorem provers are applied here.

Many model checkers for software verification use this approach.