Kapitel 3: Tools für CTL und LTL

Temporale Logik : A. Prior ~ 1960 (Philosoph)

—— " —— für (nebenläufige) Programme:

A. Pnueli 1981 (LTL)

CTL: E. Clarke, E.A. Emerson 1980-81

CTL*: E.A. Emerson, J. Halpern ~ 1986

(als Rahmen für CTL und LTL)

CTL-Model-Checking:

E. Clarke, E.A. Emerson 1981

J. Quielle, J. Sifakis 1981

LTL-Model-Checking:

M. Vardi, P. Wolper 1984

SMV - model-checker: K. McMillan 1983

Carnegie Mellon Univ.:

www-2.cs.cmu.edu/~modelcheck/code.htm

(Gruppe von E. Clarke) (nur CTL)

NuSMV (Reimplementation der Univ. Trient)

A. Cimatti, M. Roveri

http://nusmv.irst.itc.it/index.html

beinhaltet: LTL und CTL.

Cadence SMV: K. McMillan ~ 2000

Neuentwicklung: Komposition u. Abstraktion von Systemen

www-cad.eecs.berkeley.edu/~kenmcmil/

Spin*) Bell Labs

http://spinroot.com/spin/whatispin.html

besonders für verteilte Systeme entwickelt

FDR2 Model-Checker für Prozess-Algebra (CSP)

spezifizierte Systeme

www.fsel.com/software.html

MAUDE Tool auf der Basis von Rewriting-Logic

mit LTL-Model-Checker-Modul

http://maude.cs.uiuc.edu/

*) Lit zu Spin:

M. Ben-Ari: Principles of the Spin Model Checker

Springer, Berlin, 2008 T BEN

G.J. Holzmann: The Spin Model Checker

Addison-Wesley, 2004 T HOL

MAUDE http://maude.cs.uiuc.edu

SRI International Computer Science Laboratory

Department of Computer Science
University of Illinois at Urbana-Champaig

The Maude System

General Maude Information

- Maude Overview
- Rewriting Logic
- The Maude Project and Team
- Mailing Lists

Maude Documentation

- Maude Primer and Examples
- Some Papers on Maude and on Rewriting Logic
- Some Talks on Maude and on Rewriting Logic
- Maude Manual and Examples
- Roadmap and Bibliography

Maude-related Tools

- Maude Tools
- Other Tools

Obtaining and Using Maude

- Download latest version of Maude 2
- All Maude 2 versions
- Maude 2 License
- Looking for Maude 1?

MAUDE http://maude.cs.uiuc.edu/primer/

5.5 Bedingte Ersetzungstheorien

Termersetzungssysteme	*Ersetzungstheorien* *Ersetzungs-Spezifikationen*
ändern	*ändern*
Termdarstellung	*Zustand*
Semantik gleich, statisch	*Semantik verschieden, dynamisch*

Beispiel 5.36 Sei $\Sigma := \{plus, suc, null\}$, $X := \{m, n\}$ und

$$E := \{plus(null, n) \approx n, plus(suc(m), n) \approx suc(plus(m, n))\}.$$

$\mathbb{N}$
0
$n \mapsto (n+1)$
Standard

$$plus(n, m) \;=\; n + m$$

$Mod(\Sigma, V, E) = Mod(V, E) = Mod(E)$ ist die Klasse aller Modelle, die die Identitäten erfüllen. Sie enthält auch endliche Modelle.

NAT in der MAUDE-Syntax

```
fmod NAT is
 sort Nat .
 op 0 : -> Nat [ctor] .
 op suc : Nat -> Nat [ctor iter] .
 op _+_ : Nat Nat -> Nat .
 vars M N : Nat .
 eq 0 + N = N .
 eq suc(M) + N = suc(M + N) .
endfm
***Aufruf reduce suc(suc(0)) + suc(0) .
```

```
fmod NAT is
 sort Nat .
 op 0 : -> Nat [ctor] .
 op suc : Nat -> Nat [ctor iter] .
 op _+_ : Nat Nat -> Nat .
 vars M N : Nat .
 eq 0 + N = N .
 eq suc(M) + N = suc(M + N) .
endfm
***Aufruf reduce suc(suc(0)) + suc(0) .

 Maude>  reduce suc(suc(0)) + suc(0) .\\
 reduce in NAT : suc^2(0) + suc(0) .\\
 ********** equation \\ eq suc(M) + N = suc(M + N) .\\ M --> suc(0)\\ N --> suc(0)\\
 suc^2(0) + suc(0) ---> suc(suc(0) + suc(0))\\
 ********** equation \\ eq suc(M) + N = suc(M + N) .\\ M --> 0 \\ N --> suc(0) \\
 suc(0) + suc(0) ---> suc(0 + suc(0)) \\
 ********** equation \\ eq 0 + N = N .\\ N --> suc(0) \\
 0 + suc(0) ---> suc(0) \\
 rewrites: 3 in 0ms cpu (36ms real) (~ rewrites/second) \\
 result Nat: suc^3(0) \\
 Maude>
```

Beispiel 1:

```
op choice : Id Id -> Id .

rl [left] : choice(x,y) => x .
rl [right] : choice(x,y) => y .
```

Beispiel 2:

```
mod lts01 is
  sort state .
  ops   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10  : -> state .

      rl [t1]: s1 => s3 .
      rl [t2]: s2 => s3 .
      rl [t3]: s2 => s9 .
      rl [t4]: s3 => s4 .
      rl [t5]: s5 => s6 .
      rl [t6]: s5 => s7 .
      rl [t7]: s6 => s8 .
      rl [t8]: s9 => s10 .
  endm
  *** Aufruf: rewrite s1 .

   result state: s4.
```

`reduce s4` ergibt `result state: s5.`

```
Aufruf: rewrite s1 .
        result state: s8.
```

Identität $s4 \approx s5$

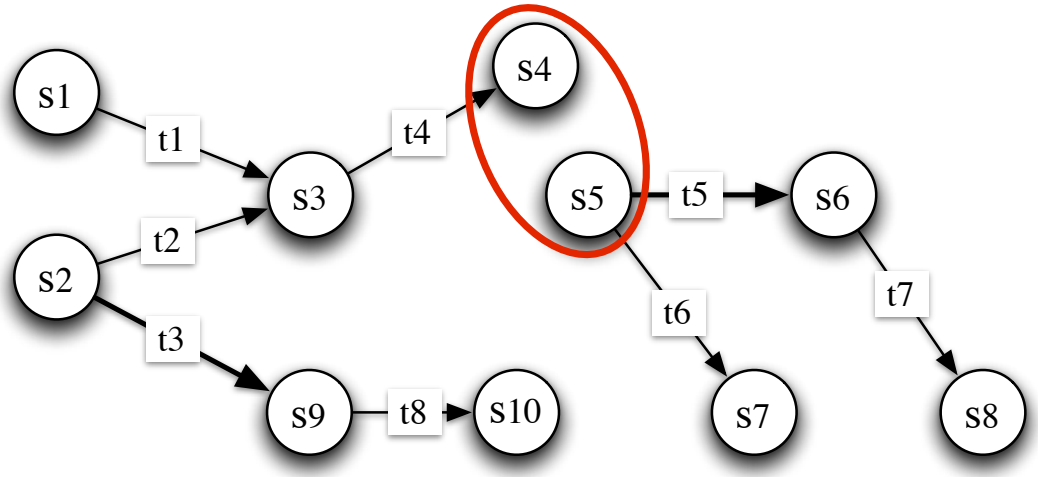


Abbildung 5.6: Ein Transitionssystem LTS

```
mod lts03 is
  sort state .
  ops   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 : -> state .
    rl [t1]: s1 => s3 .
    rl [t2]: s2 => s9 .
    rl [t3]: s2 => s3 .
    crl [t4]: s3 => s4 if s2 => s10 /\ s6 => s8.
    rl [t5]: s3 => s9 .
    rl [t6]: s5 => s6 .
    rl [t7]: s5 => s7 .
    rl [t8]: s6 => s8 .
    rl [t9]: s9 => s10 .
 eq s4 = s5 .
endm\label{lts03}
```

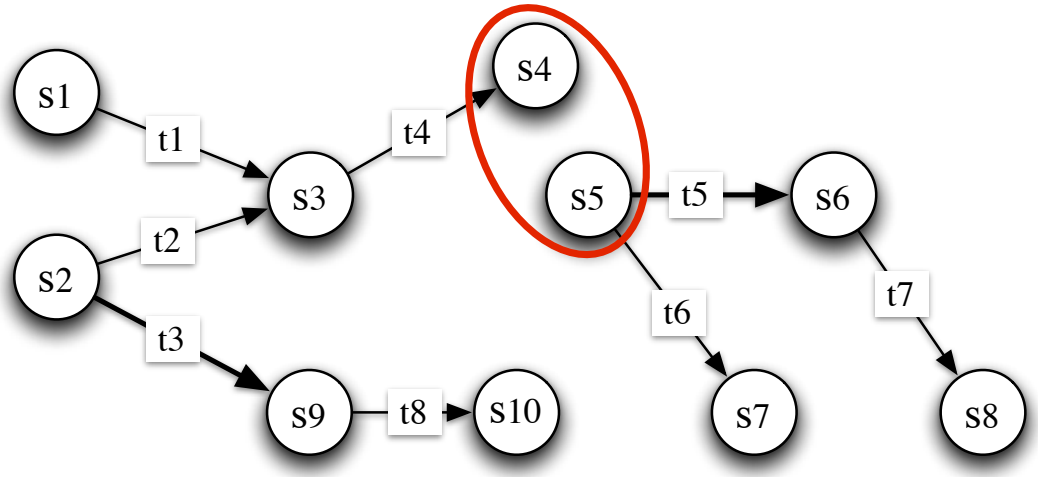


Abbildung 5.6: Ein Transitionssystem LTS

```
mod lts03 is
  sort state .
  ops   s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 : -> state .
    rl [t1]: s1 => s3 .
    rl [t2]: s2 => s9 .
    rl [t3]: s2 => s3 .
    crl [t4]: s3 => s4 if s2 => s10 /\ s6 => s8.
    rl [t5]: s3 => s9 .
    rl [t6]: s5 => s6 .
    rl [t7]: s5 => s7 .
    rl [t8]: s6 => s8 .
    rl [t9]: s9 => s10 .
 eq s4 = s5 .
endm\label{lts03}
```

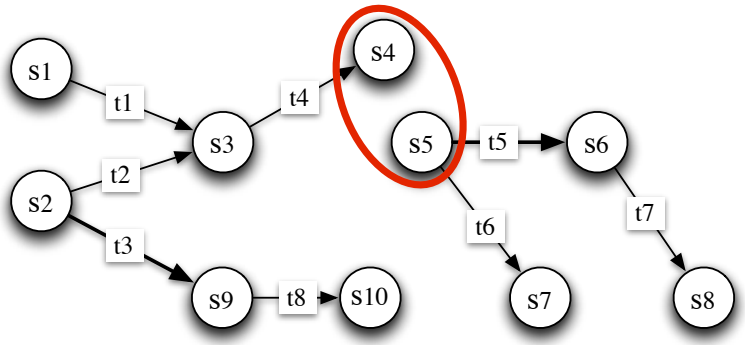


Abbildung 5.6: Ein Transitionssystem LTS

```
Maude> rewrite s1 . \\ rewrite in lts03 : s1 .
*********** rule \\ rl s1 => s3 [label t1] .\\ s1 ---> s3 \\
*********** trial #1 \\ crl s3 => s4 if s2 => s10 /\ s6 => s8 [label t4] .\\
*********** solving condition fragment \\ s2 => s10 \\
*********** rule \\ rl s2 => s9 [label t2] .\\ s2 ---> s9 \\
*********** rule \\ rl s2 => s3 [label t3] .\\ s2 ---> s3 \\
*********** rule \\rl s9 => s10 [label t9] .\\ s9 ---> s10 \\
*********** success for condition fragment \\ s2 => s10 \\
*********** solving condition fragment \\ s6 => s8 \\
*********** rule \\ rl s6 => s8 [label t8] .\\ s6 ---> s8 \\
*********** success for condition fragment \\ s6 => s8 \\
*********** success #1 \\
*********** rule\\crl s3 => s4 if s2 => s10 /\ s6 => s8 [label t4].\\ s3--->s4 \\
*********** equation \\ eq s4 = s5 .\\ s4 ---> s5 \\
*********** rule \\ rl s5 => s6 [label t6] .\\ s5 ---> s6 \\
*********** rule \\ rl s6 => s8 [label t8] .\\ s6 ---> s8 \\
rewrites: 9 in 20ms cpu (219ms real) (450 rewrites/second) \\
result state: s8 \\ Maude>
```

5.5.2 Eine formale Definition

Definition 5.68 *Eine* bedingte Ersetzungstheorie $\mathcal{R}$ *ist das Tupel*

$$\mathcal{R} = (\mathcal{D}, L, R)$$

- $\mathcal{D} = (\Sigma, X, E)$ *ist eine Spezifikation.*
- L *ist eine Menge von Labeln.*
- *Die Menge der Ersetzungsregeln* R *ist eine Teilmenge von Paaren der Form:*

  $$R \subseteq L \times (\mathbb{T}_{\Sigma,E}(X)^2)^+$$

  Die erste Komponente ist ein Label und die zweite ist eine nichtleere Sequenz von Paaren der E*-Äquivalenzklassen.*

Für eine Regel der Form $(l, ([t], [t'])([u_1], [v_1]) \ldots ([u_k], [v_k])) \in R$ wird auch die folgende Notation verwendet:

$$l : [t] \to [t'] \text{ if } [u_1] \to [v_1] \wedge \ldots \wedge [u_k] \to [v_k]$$

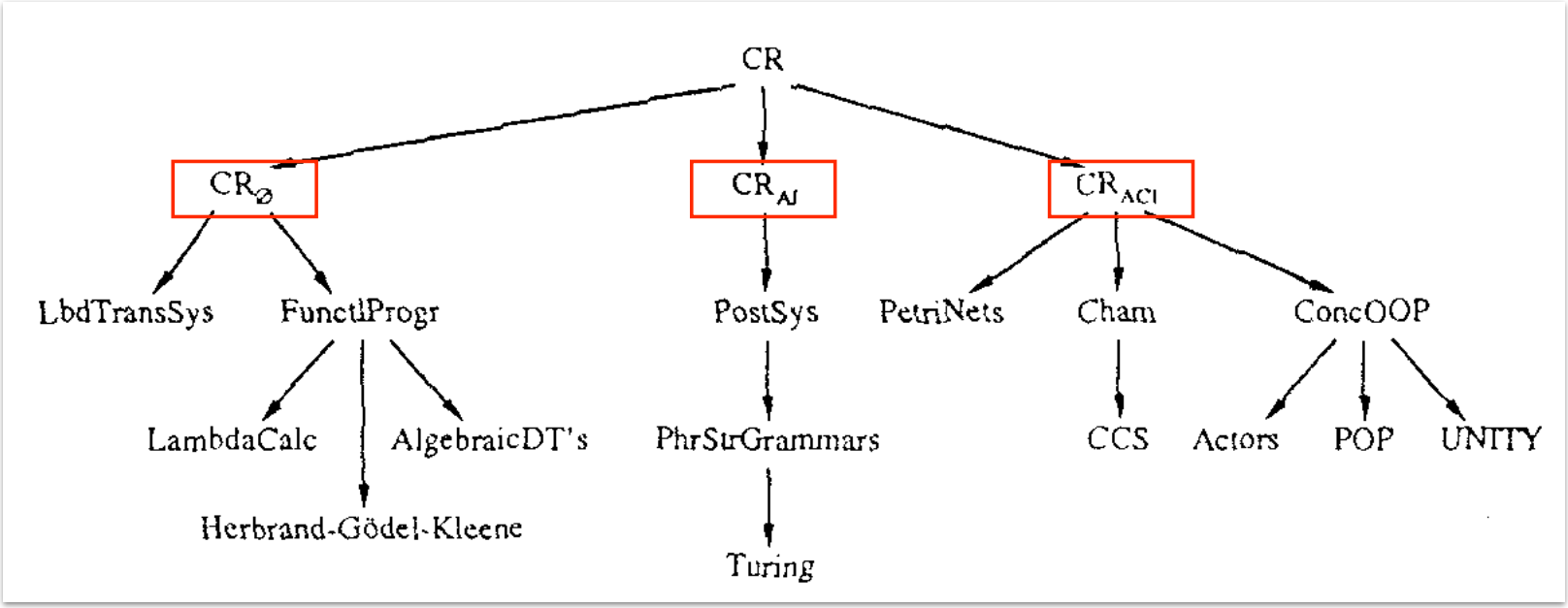
Deduktion In einer Ersetzungstheorie $\mathcal{R}$ ist eine Sequenz $\mathcal{R} \vdash [t] \to [t']$ genau dann ableitbar, wenn sie durch endliche Anwendung der folgenden *Deduktionsregeln* erzeugbar ist:

(a) *Reflexivität.* Für jedes $[t] \in \mathbb{T}_{\Sigma,E}(X)$ existiert die Regel:

$$\frac{}{[t] \to [t]}$$

(b) Σ-*Struktur.* Für jedes Funktionssymbol $f \in k_1 \times \cdots \times k_n \to k$ und Terme $t_i, t_i' \in \mathbb{T}_{\Sigma,E}^{k_i}(X)$ für $1 \leq i \leq n$ existiert die Regel:

$$\frac{[t_1] \to [t_1'] \ldots [t_n] \to [t_n']}{[f(t_1, \ldots, t_n)] \to [f(t_1', \ldots, t_n')]}$$



Hauptklassen:

$CR_{\emptyset}$ CR_{AI} CR_{ACI}

$E = \{[\texttt{assoc}], [\texttt{id}]\}$

$E = \emptyset$

$E = \{[\texttt{assoc}], [\texttt{comm}], [\texttt{id}]\}$

CR_{AI}

$E = \{$[assoc], [id]$\}$

```
mod acceptor is
 sorts letter word .
 subsort letter < word .
  ops A B a b c # akzeptiert : -> letter .
  op empty : -> letter .
  op __ : word word -> word  [assoc id: empty] .

rl [0] : # # =>  akzeptiert .
rl [1]  : # a  => #  A .
rl [2]  : A a => a A .
rl [3]  : A b => B .
rl [4]  : B b => b B .
rl [5]  : B c => empty .
endm
```

```
***Aufruf rew  # a a a a b b b b c c c c #  .

***Aufruf rew  # a a a b b b b c c c c #  .
```

Beispiel: Mengen

```
sort IdSet .
subsort Id < IdSet .

vars s s' s'' : IdSet .

op emptyIdSet : -> IdSet .
op idSet : IdSet IdSet -> IdSet
   [assoc comm] .

eq idSet(s,s) = s .
eq idSet(emptyIdSet,s) = s .

op inIdSet : Id IdSet -> Bool .
eq inIdSet(x,emptyIdSet) = false .
eq inIdSet(x,idSet(x',s')) =
   if x == x' then true else inIdSet(x,s') fi .
```

Mengenvereinigung

```
sort IdSet .
subsort Id < IdSet .

vars s s' s'' : IdSet .

op emptyIdSet : -> IdSet .
op idSet : IdSet IdSet -> IdSet
   [assoc comm] .

eq idSet(s,s) = s .
eq idSet(emptyIdSet,s) = s .

op inIdSet : Id IdSet -> Bool .
eq inIdSet(x,emptyIdSet) = false .
eq inIdSet(x,idSet(x',s')) =
   if x == x' then true else inIdSet(x,s') fi .
```

Beispiel: Multimengen

`f(f(f(x1,x2),x3))` als `f(x1,x2,x3)`.

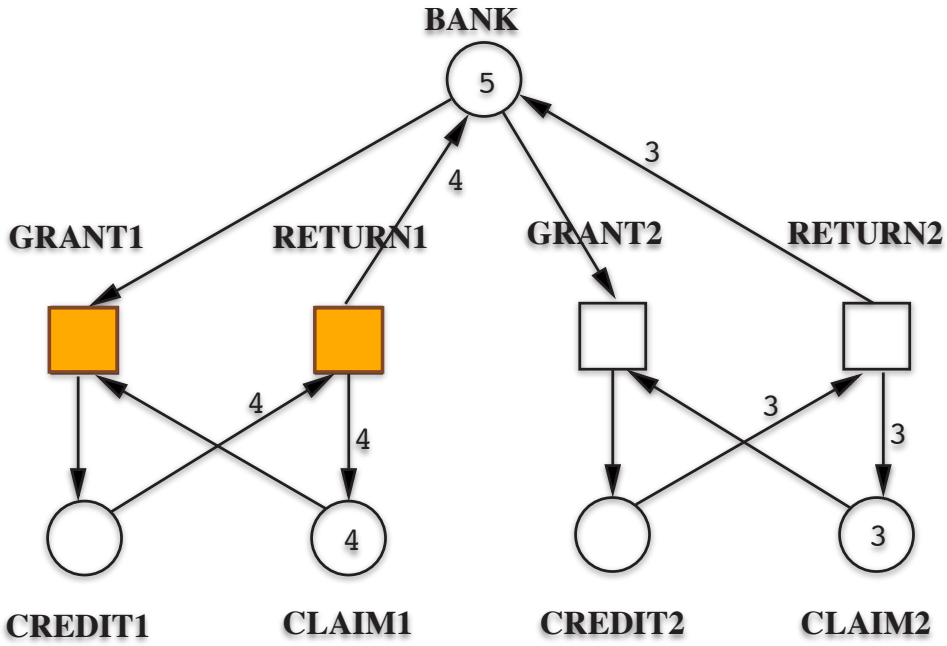
`idBag(a,b,b,a,d)`

```
sort IdBag .

subsort Id < IdBag .

op emptyIdBag : -> IdBag .
op idBag : IdBag IdBag -> IdBag
   [assoc comm id: emptyIdBag] .
```

Multimengen-Vereinigung

CR_{ACI}

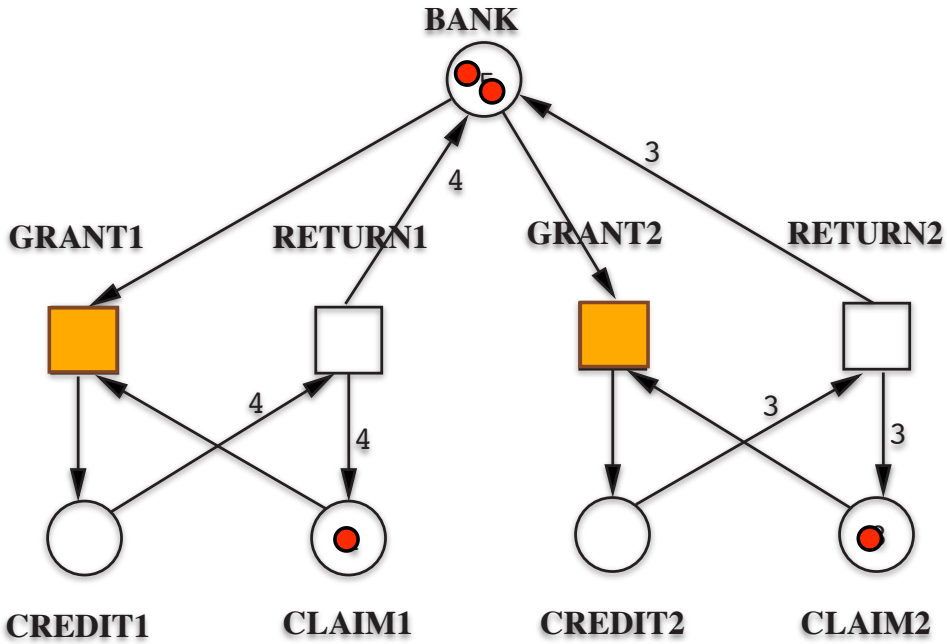
$E = \{[\texttt{assoc}], [\texttt{comm}], [\texttt{id}]\}$




```
mod banker is
 sorts Token Marking .
subsort Token < Marking .

op emptyMarking : -> Marking .
op __ : Marking Marking -> Marking  [assoc comm id: emptyMarking] .
     rl [GRANT1]  : BANK CLAIM1 => CREDIT1 .
     rl [RETURN1] : CREDIT1 CREDIT1 CREDIT1 CREDIT1  =>
                    BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1  .
     rl [GRANT2]  : BANK CLAIM2  => CREDIT2 .
     rl [RETURN2] : CREDIT2 CREDIT2 CREDIT2  =>
                    BANK BANK BANK CLAIM2 CLAIM2 CLAIM2  .
     endm

Aufruf rewrite BANK BANK CLAIM1 CLAIM2   .
```

CR_{ACI}

$E = \{[\texttt{assoc}], [\texttt{comm}], [\texttt{id}]\}$



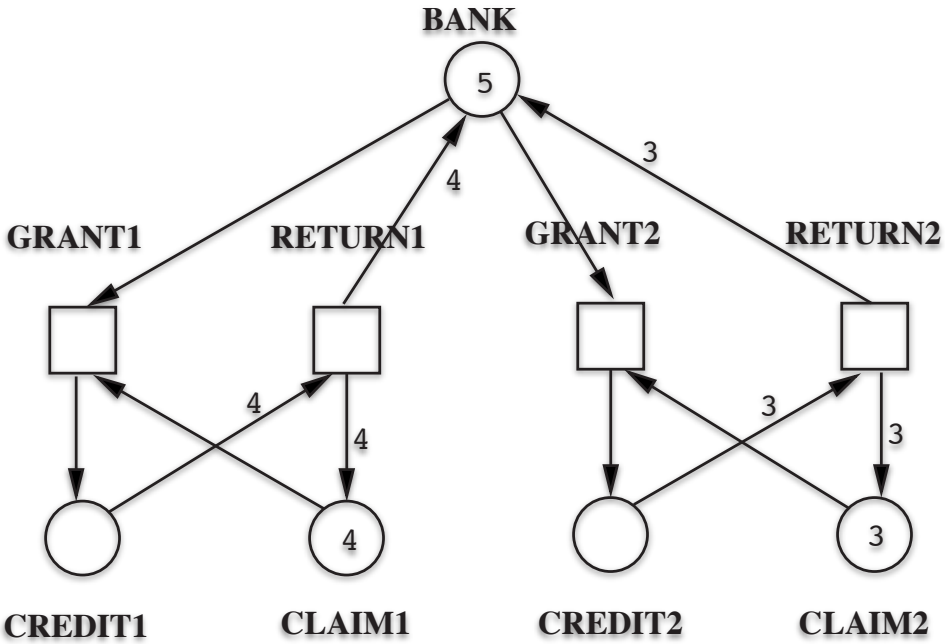

```
mod banker is
 sorts Token Marking .
subsort Token < Marking .

op emptyMarking : -> Marking .
op __ : Marking Marking -> Marking  [assoc comm id: emptyMarking] .
     rl [GRANT1]  : BANK CLAIM1 => CREDIT1 .
     rl [RETURN1] : CREDIT1 CREDIT1 CREDIT1 CREDIT1  =>
                    BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1  .
     rl [GRANT2]  : BANK CLAIM2  => CREDIT2 .
     rl [RETURN2] : CREDIT2 CREDIT2 CREDIT2  =>
                    BANK BANK BANK CLAIM2 CLAIM2 CLAIM2  .
     endm
```

Aufruf rewrite BANK BANK CLAIM1 CLAIM2 .

result Marking: CREDIT1 CREDIT2

CR_{ACI}

$E = \{[\texttt{assoc}], [\texttt{comm}], [\texttt{id}]\}$




```
mod banker is
 sorts Token Marking .
subsort Token < Marking .

op emptyMarking : -> Marking .
op __ : Marking Marking -> Marking  [assoc comm id: emptyMarking] .
     rl [GRANT1]  : BANK CLAIM1 => CREDIT1 .
     rl [RETURN1] : CREDIT1 CREDIT1 CREDIT1 CREDIT1  =>
                    BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1  .
     rl [GRANT2]  : BANK CLAIM2  => CREDIT2 .
     rl [RETURN2] : CREDIT2 CREDIT2 CREDIT2  =>
                    BANK BANK BANK CLAIM2 CLAIM2 CLAIM2  .
     endm
***Aufruf rewrite[10]
BANK BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1 CLAIM2 CLAIM2 CLAIM2   .
                  result Marking: CREDIT1 CREDIT1 CREDIT1 CREDIT2 CREDIT2 CLAIM1 CLAIM2
```

The Maude LTL Model Checker and Its Implementation*

T. Ball and S. K. Rajamani (Eds.): SPIN 2003, LNCS 2648, pp. 230–234, 2003

Steven Eker[1], José Meseguer[2], and Ambarish Sridharanarayanan[2]

Maude specifications are *executable logical theories* in rewriting logic [9], a logic that is a flexible logical framework for expressing a very wide range of concurrency models and distributed systems [9]. A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational theory specifying a system's *distributed state structure* and with R a collection of rewrite rules specifying the *concurrent transitions* of the system. Since no domain-specific model of concurrency is built into the logic, the range of applications that can be naturally specified is indeed very wide. For example, besides conventional distributed system specifications, properties of signalling pathways in mammalian cells have been model checked [3]. Another advantage of Maude as the system specification language is that integration of model checking with theorem proving techniques becomes quite seamless. The same rewrite theory $\mathcal{R} = (\Sigma, E, R)$ can be the input to the LTL model checker and to several other proving tools in the Maude environment [2]. For a lengthier discussion of the Maude LTL model checker and its LTL satisfiability and tautology procedures, see the companion paper [4].

Then, given an initial state, say `init` of sort $\texttt{State}_{\texttt{M}}$, we can *model check* different LTL properties beginning at this initial state by doing the following:

- defining a new module, say `CHECK-M`, that includes the modules `M` and the predefined module `MODEL-CHECKER` as submodules;
- giving a *subsort declaration*, `subsort` $\texttt{State}_{\texttt{M}}$ `< State .`, where `State` is one of the key sorts in the module `MODEL-CHECKER`;
- defining the *syntax* of the *state predicates* we wish to use (which can be parametric) by means of constants and operators of sort `Prop`, a subsort of the sort `Formula` (i.e., LTL formulas) in the module `MODEL-CHECKER`;
- defining the *semantics* of the state predicates by means of equations.

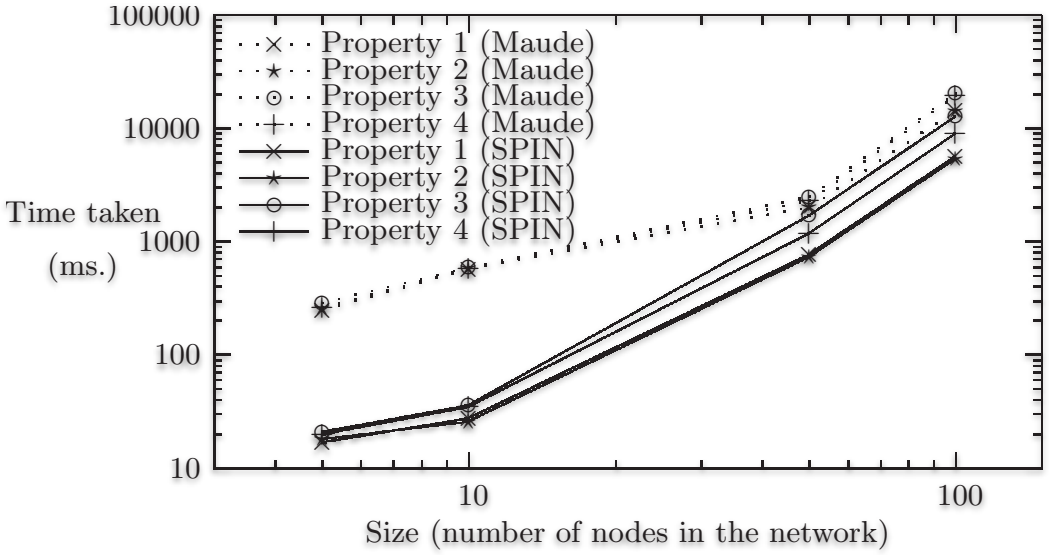
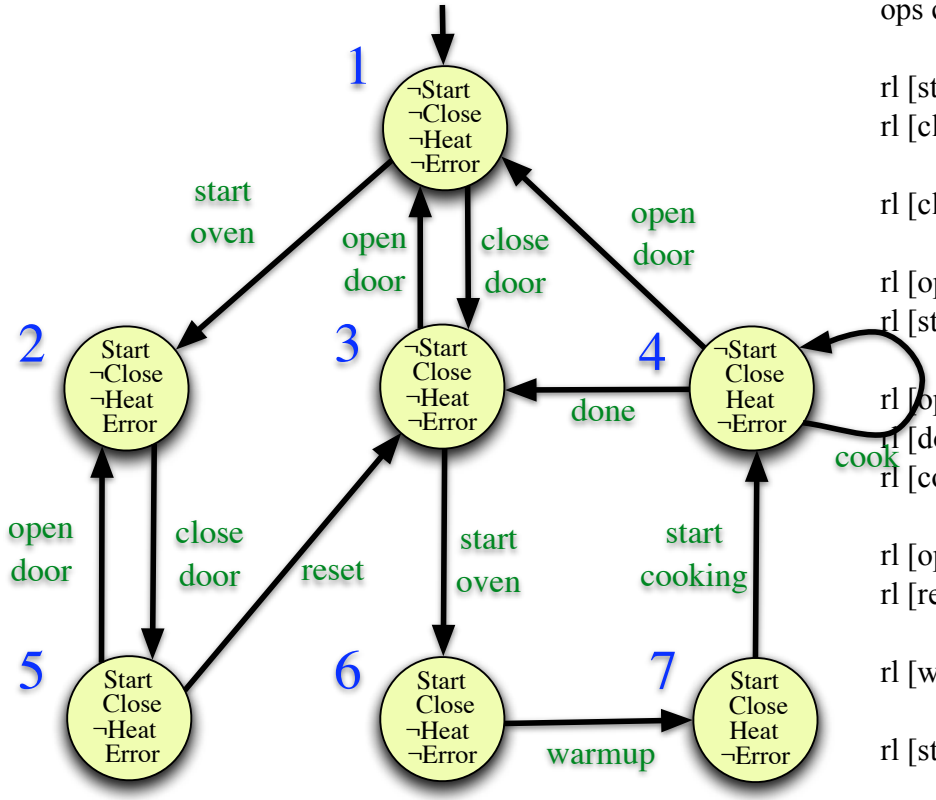


Fig. 1. Execution time for the leader-election problem - Maude vs. SPIN

$$f = AG(Start \rightarrow AF\ Heat)$$



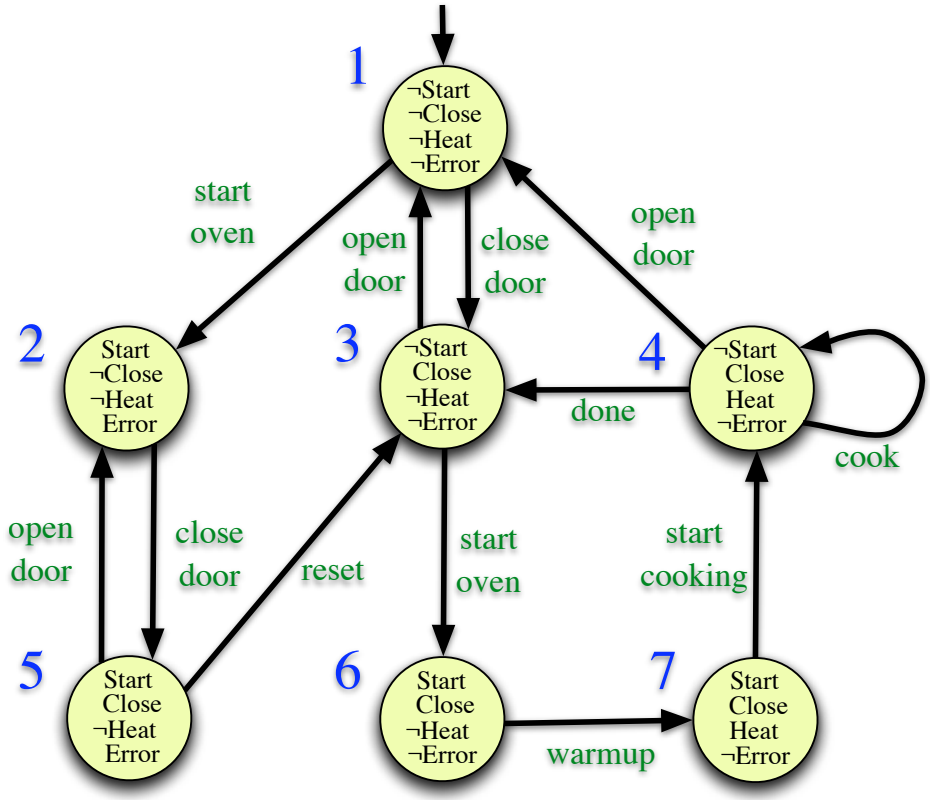

```
mod OFEN5 is

inc MODEL-CHECKER .

ops s1 s2 s3 s4 s5 s6 s7 : -> State .
ops close heat start error :  -> Prop

rl [start_oven]: s1 => s2 .
rl [close_door]: s1 => s3 .

rl [close_door]: s2 => s5 .

rl [open_door]: s3 => s1 .
rl [start_oven]: s3 => s6 .

rl [open_door]: s4 => s1 .
rl [done]: s4 => s3 .
rl [cook]: s4 => s4 .

rl [open_door]: s5 => s2 .
rl [reset]: s5 => s3 .

rl [warmup]: s6 => s7 .

rl [start_cooking]: s7 => s4 .
```

```
*** Nur die postiven Fälle sind zu
*** spezifizieren.
eq ( s2  |= start)  = true .
eq ( s2  |= error ) = true .
*****************
eq ( s3  |= close)  = true .
*****************
eq ( s4  |= close)  = true .
eq ( s4  |= heat )  = true .
*****************
eq ( s5  |= start)  = true .
eq ( s5  |= close)  = true .
eq ( s5  |= error ) = true .
*****************
eq ( s6  |= start)  = true .
eq ( s6  |= close)  = true .
*****************
eq ( s7  |= start)  = true .
eq ( s7  |= close)  = true .
eq ( s7  |= heat )  = true .
*****************

endm
```

```
red modelCheck(s1, [] (start  -> <> heat)) .
```

$$f = AG(Start \rightarrow AF\ Heat)$$

G (start -> F heat)




```
Last login: Sat Apr 26 15:14:35 on ttys000
[PB-G4-V-1:-] rvalk% maude.darwin
                     \|||||||||||||||||/
                   --- Welcome to Maude ---
                     /|||||||||||||||||\
          Maude alpha85a built: Dec 23 2004 17:01:06
          Copyright 1997-2004 SRI International
                Sat Apr 26 15:17:28 2008
Maude> in model-checker.maude
==========================================
fmod LTL
==========================================
fmod LTL-SIMPLIFIER
==========================================
fmod SAT-SOLVER
==========================================
fmod SATISFACTION
==========================================
fmod MODEL-CHECKER
Maude>
> in /Users/rvalk/Desktop/MC-Modelle/10fen.maude
==========================================
mod OFEN5
Maude> red modelCheck(s1, [] (start  -> <> heat)) .
reduce in OFEN5 : modelCheck(s1, [](start -> <> heat)) .
rewrites: 17 in 2ms cpu (3ms real) (6172 rewrites/second)
result ModelCheckResult: counterexample({s1,'start_oven},
 {s2,'close_door} {s5,'reset} {s3,'open_door} {s1,'start_oven})
Maude>
```

```
mod banker is
inc MODEL-CHECKER .
sorts Token Marking .
subsort Token < Marking .


op emptyMarking : -> Marking .
op __ : Marking Marking -> Marking  [assoc comm id: emptyMarking] .

var m : Marking .

op BANK : -> Token .
op CREDIT1 : -> Token .
op CREDIT2 : -> Token .
op CLAIM1 : -> Token .
op CLAIM2 : -> Token .

rl [GRANT1]  : BANK CLAIM1 => CREDIT1 .

rl [RETURN1] : CREDIT1 CREDIT1 CREDIT1 CREDIT1  =>
               BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1  .

rl [GRANT2]  : BANK CLAIM2  => CREDIT2 .

rl [RETURN2] : CREDIT2 CREDIT2 CREDIT2  =>   BANK BANK BANK CLAIM2 CLAIM2 CLAIM2  .

op initial : -> Marking .
eq initial = BANK BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1 CLAIM2 CLAIM2 CLAIM2 .
```

```
*** Überdecken von #BANK #CREDIT1 #CLAIM1 #CREDIT2 #CLAIM2
ops cov  :  Nat Nat Nat Nat Nat -> Prop .
eq ( m  |= cov(0,0,0,0,0)  )  = true .
eq ( m  BANK |= cov(s n1,n2,n3,n4,n5) )  = ( m  |= cov(n1,n2,n3,n4,n5) ) .
eq ( m  CREDIT1 |= cov(n1,s n2,n3,n4,n5) )  = ( m  |= cov(n1,n2,n3,n4,n5) ) .
eq ( m  CLAIM1 |= cov(n1,n2,s n3,n4,n5) )  = ( m  |= cov(n1,n2,n3,n4,n5) ) .
eq ( m  CREDIT2 |= cov(n1,n2,n3,s n4,n5) )  = ( m  |= cov(n1,n2,n3,n4,n5) ) .
eq ( m  CLAIM2 |= cov(n1,n2,n3,n4,s n5) )  = ( m  |= cov(n1,n2,n3,n4,n5) ) .

endm

*** Aufruf:  red modelCheck(initial, <> cov(5,0,4,0,3)) .
*** Aufruf:  red modelCheck(initial, <> cov(6,0,4,0,3)) .

*** Aufruf:  red modelCheck(initial, []<> ( cov(1,0,1,0,0) \/ cov(0,4,0,0,0) \/
cov(1,0,0,1,0)\/ cov(1,0,0,3,0)  )) .

result ModelCheckResult: counterexample({BANK BANK BANK BANK BANK CLAIM1 CLAIM1 CLAIM1 CLAIM1
CLAIM2 CLAIM2 CLAIM2,'GRANT1} {BANK BANK
    BANK BANK CREDIT1 CLAIM1 CLAIM1 CLAIM1 CLAIM2 CLAIM2 CLAIM2,'GRANT1} {BANK BANK BANK
CREDIT1 CREDIT1 CLAIM1 CLAIM1 CLAIM2 CLAIM2
    CLAIM2,'GRANT1} {BANK BANK CREDIT1 CREDIT1 CREDIT1 CLAIM1 CLAIM2 CLAIM2 CLAIM2,'GRANT1}
{BANK CREDIT1 CREDIT1 CREDIT1 CREDIT1 CLAIM2
    CLAIM2 CLAIM2,'GRANT2} {CREDIT1 CREDIT1 CREDIT1 CREDIT1 CREDIT2 CLAIM2 CLAIM2,'RETURN1}
{BANK BANK BANK BANK CREDIT2 CLAIM1 CLAIM1
    CLAIM1 CLAIM1 CLAIM2 CLAIM2,'GRANT1} {BANK BANK BANK CREDIT1 CREDIT2 CLAIM1 CLAIM1 CLAIM1
CLAIM2 CLAIM2,'GRANT1} {BANK BANK CREDIT1
    CREDIT1 CREDIT2 CLAIM1 CLAIM1 CLAIM2 CLAIM2,'GRANT1} {BANK CREDIT1 CREDIT1 CREDIT1
CREDIT2 CLAIM1 CLAIM2 CLAIM2,'GRANT2}, {CREDIT1
    CREDIT1 CREDIT1 CREDIT2 CREDIT2 CLAIM1 CLAIM2,deadlock})
Maude>
```

Alternierbitprotokoll-kurz

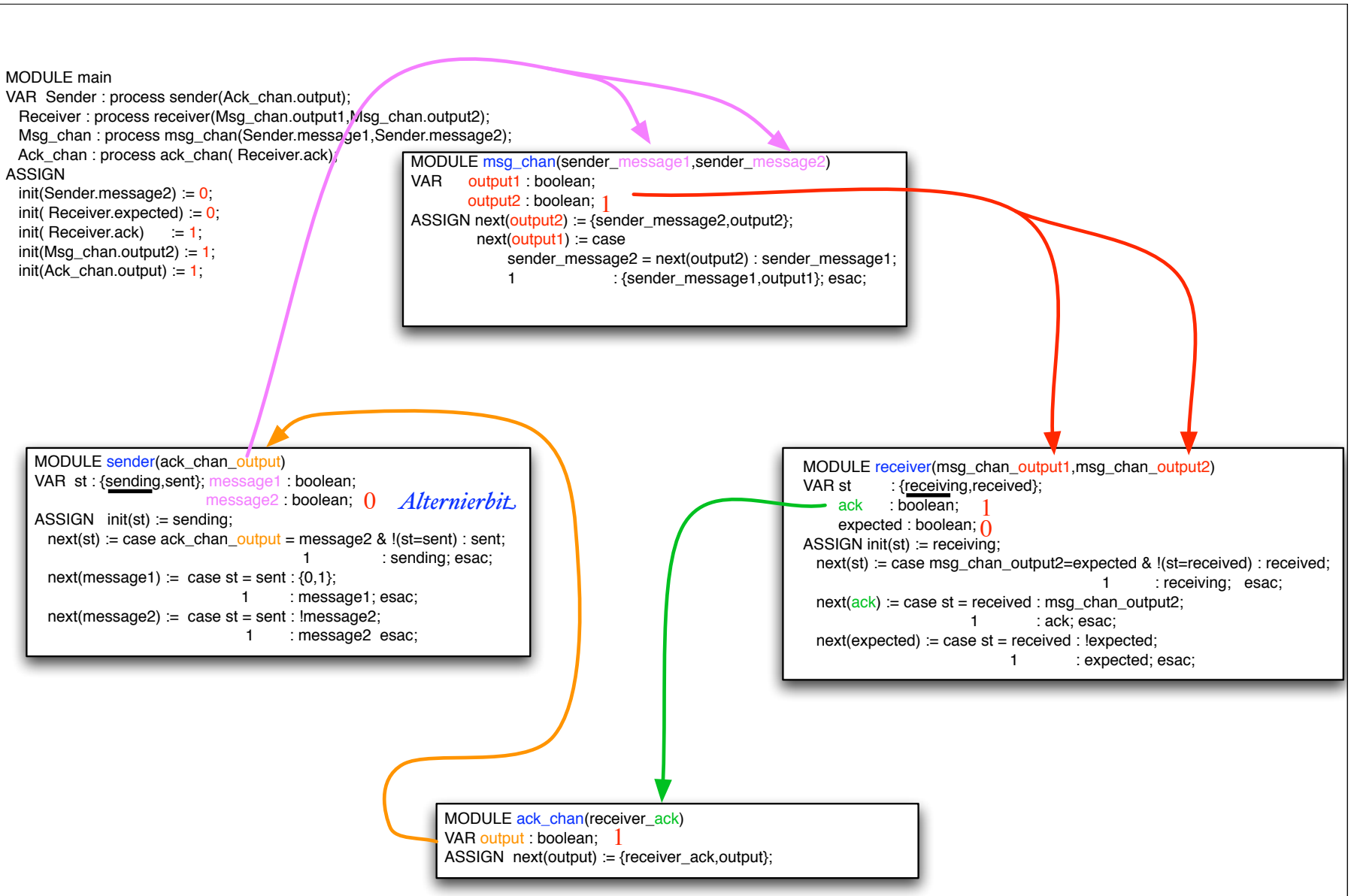
```
MODULE main
VAR  Sender : process sender(Ack_chan.output);
   Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
   Msg_chan : process msg_chan(Sender.message1,Sender.message2);
   Ack_chan : process ack_chan( Receiver.ack);
ASSIGN
   init(Sender.message2) := 0;
   init( Receiver.expected) := 0;
   init( Receiver.ack)      := 1;
   init(Msg_chan.output2) := 1;
   init(Ack_chan.output) := 1;
```

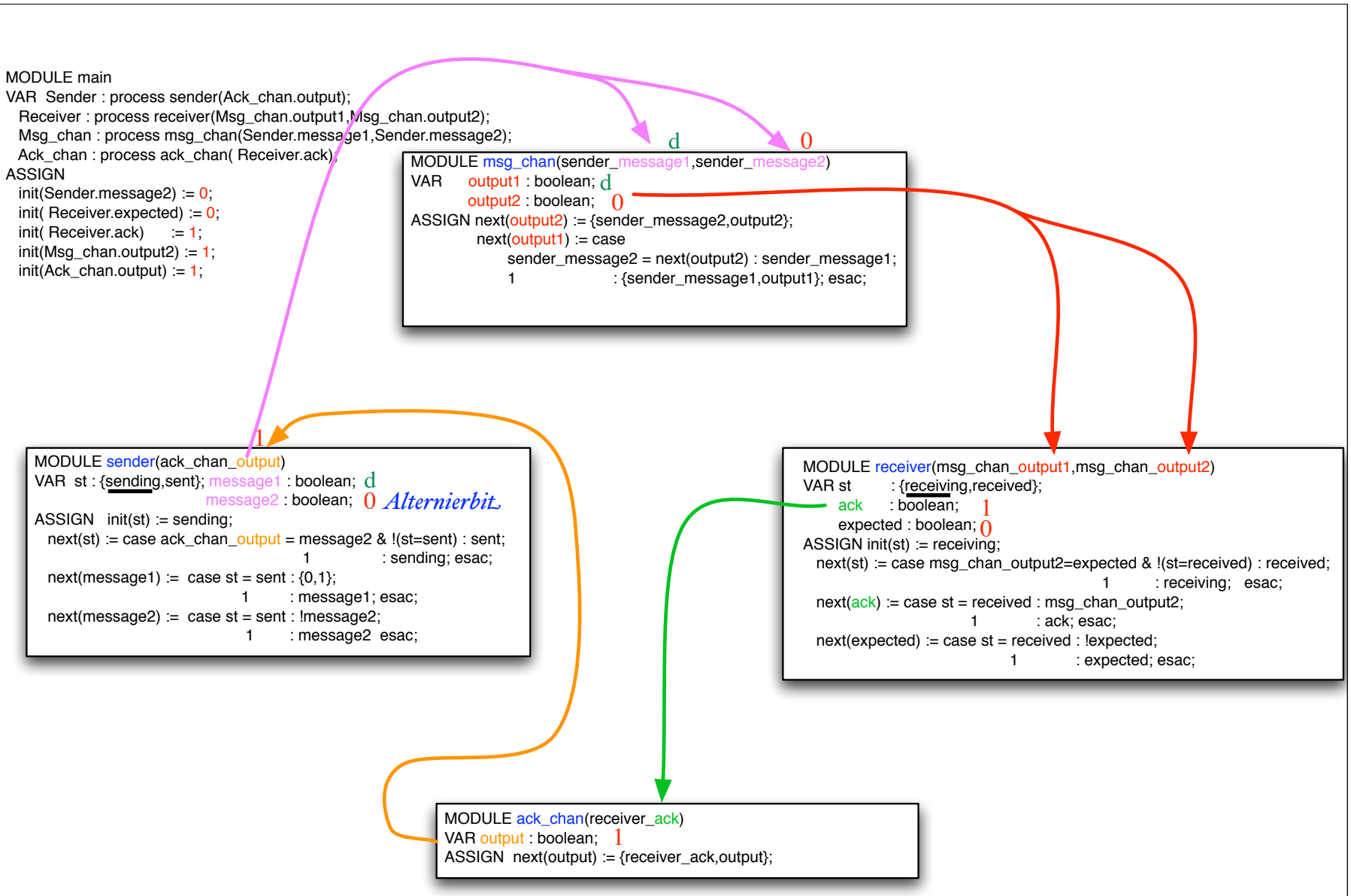
```
MODULE msg_chan(sender_message1,sender_message2)
VAR      output1 : boolean;
         output2 : boolean;
ASSIGN next(output2) := {sender_message2,output2};
          next(output1) := case
               sender_message2 = next(output2) : sender_message1;
               1                  : {sender_message1,output1}; esac;
```

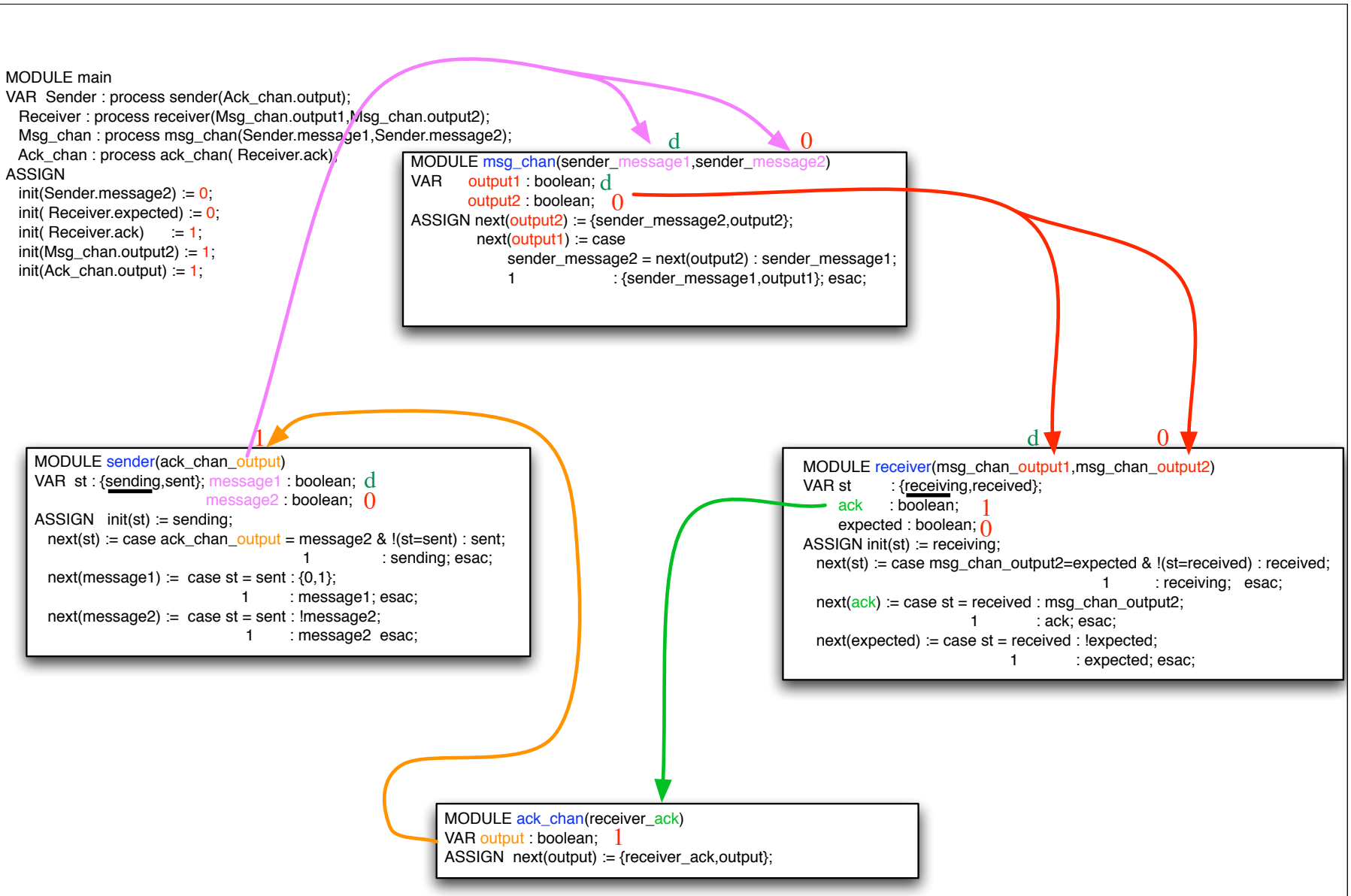
```
MODULE sender(ack_chan_output)
VAR  st : {sending,sent}; message1 : boolean;
                          message2 : boolean;   Alternierbit
ASSIGN   init(st) := sending;
  next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
                                   1               : sending; esac;
  next(message1) :=  case st = sent : {0,1};
                              1        : message1; esac;
  next(message2) :=  case st = sent : !message2;
                               1       : message2  esac;
```

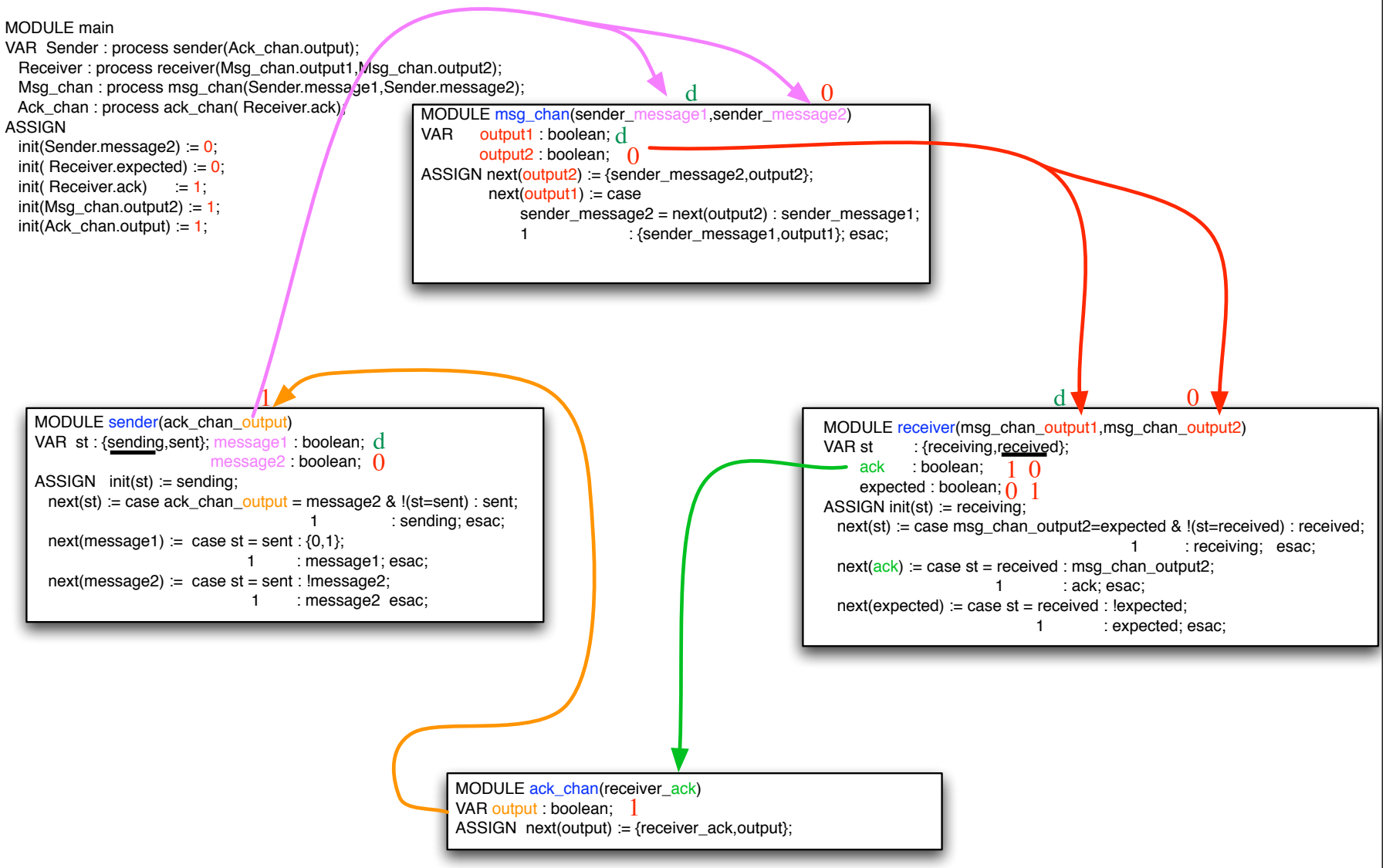
```
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st        : {receiving,received};
    ack       : boolean;
    expected : boolean;
ASSIGN init(st) := receiving;
  next(st) := case msg_chan_output2=expected & !(st=received) : received;
                                                  1         : receiving;   esac;
  next(ack) := case st = received : msg_chan_output2;
                             1            : ack; esac;
  next(expected) := case st = received : !expected;
                                1            : expected; esac;
```

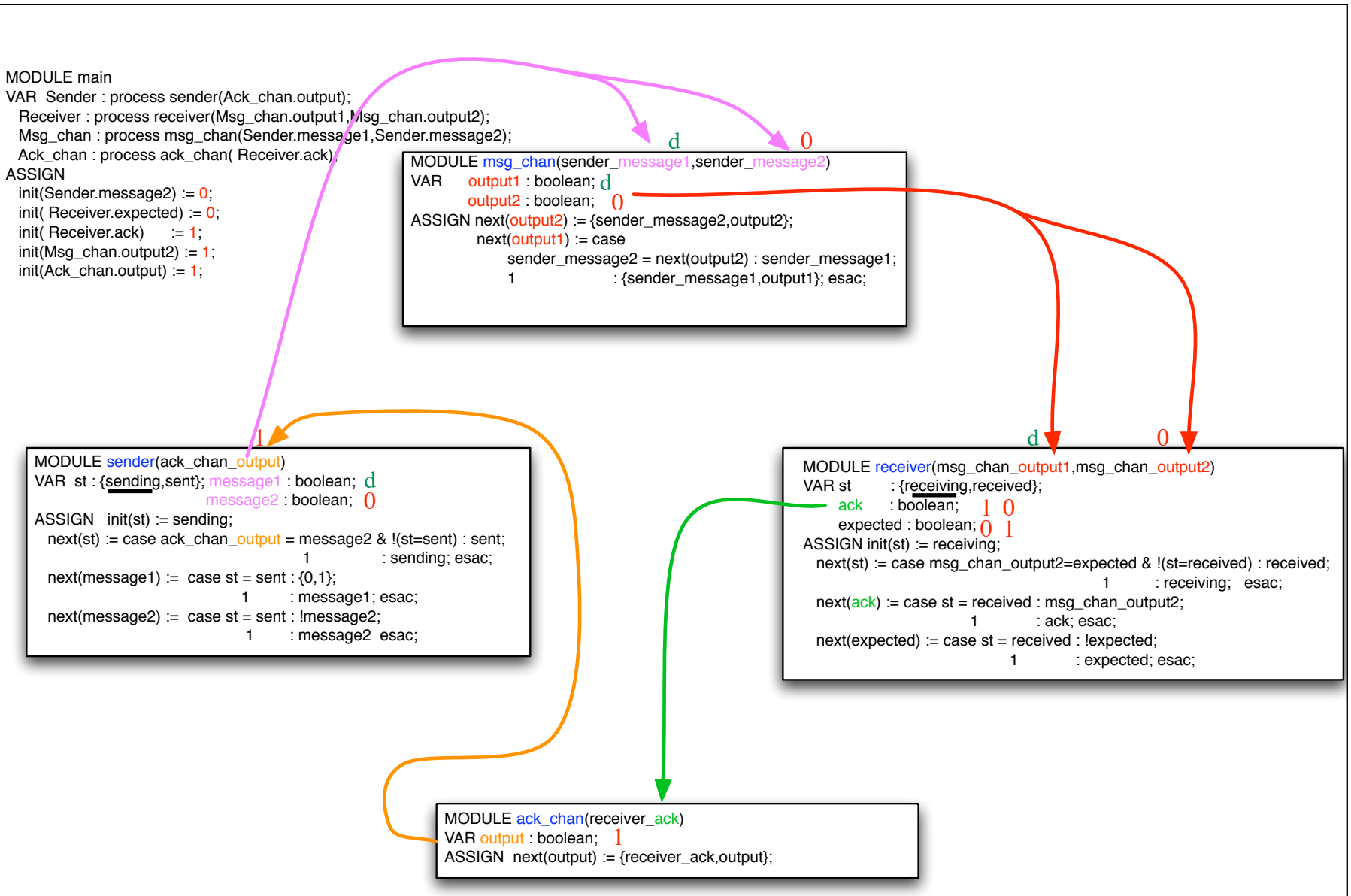
```
MODULE ack_chan(receiver_ack)
VAR output : boolean;
ASSIGN  next(output) := {receiver_ack,output};
```

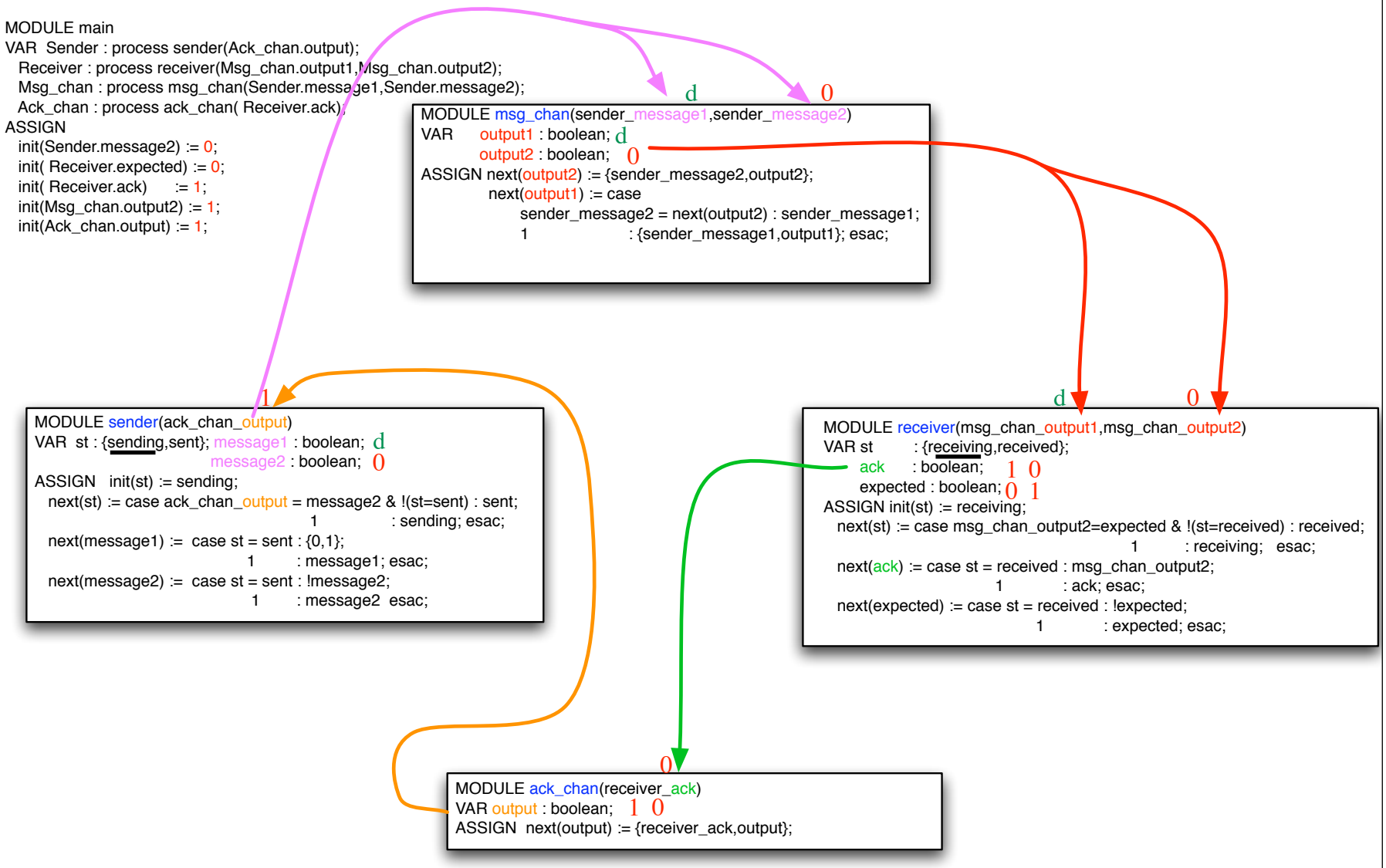

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean;
output2 : boolean; 1
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean;
message2 : boolean; 0
Alternierbit
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1
expected : boolean; 0
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1
ASSIGN next(output) := {receiver_ack,output};

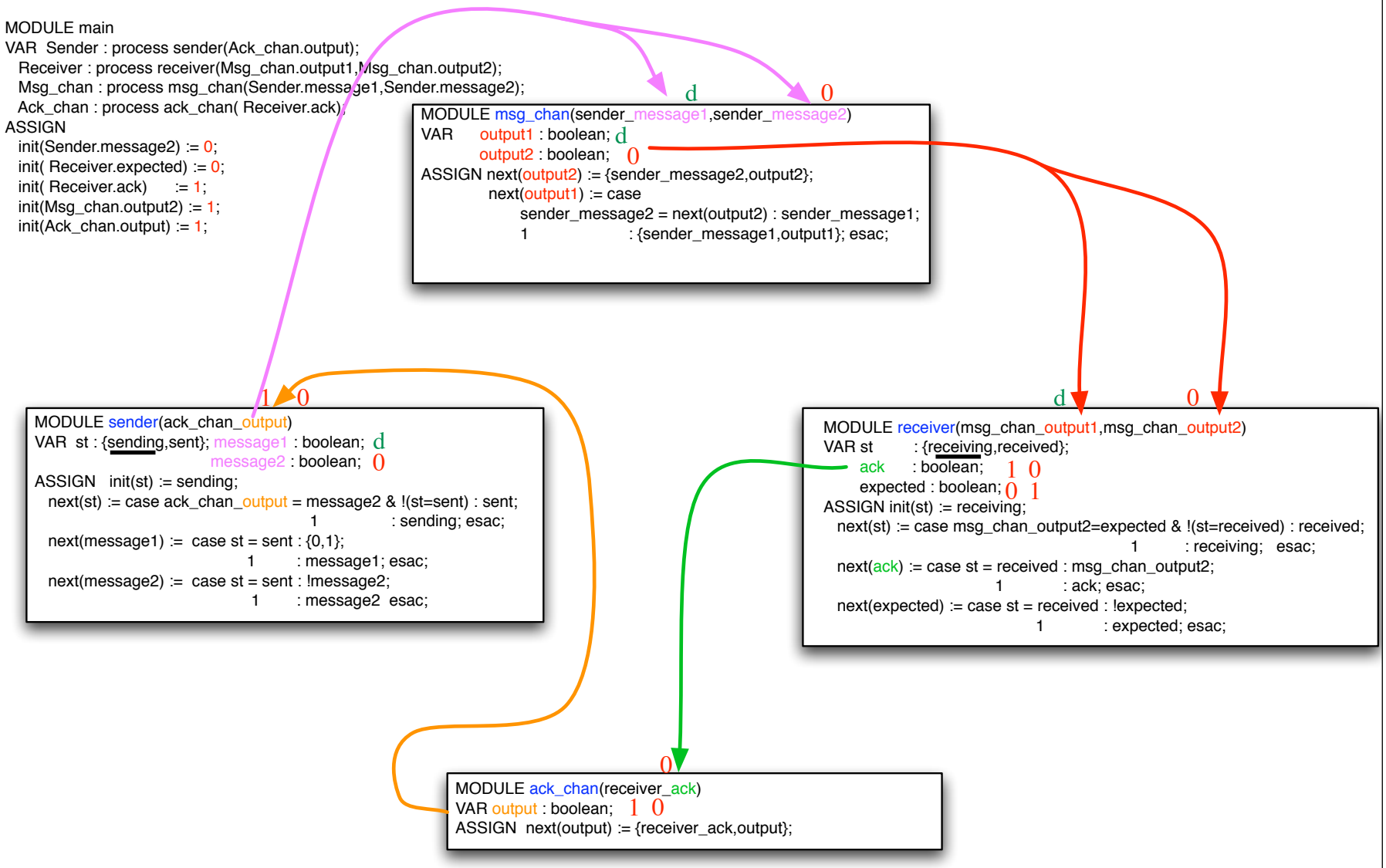

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0 Alternierbit
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1
expected : boolean; 0
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1
ASSIGN next(output) := {receiver_ack,output};

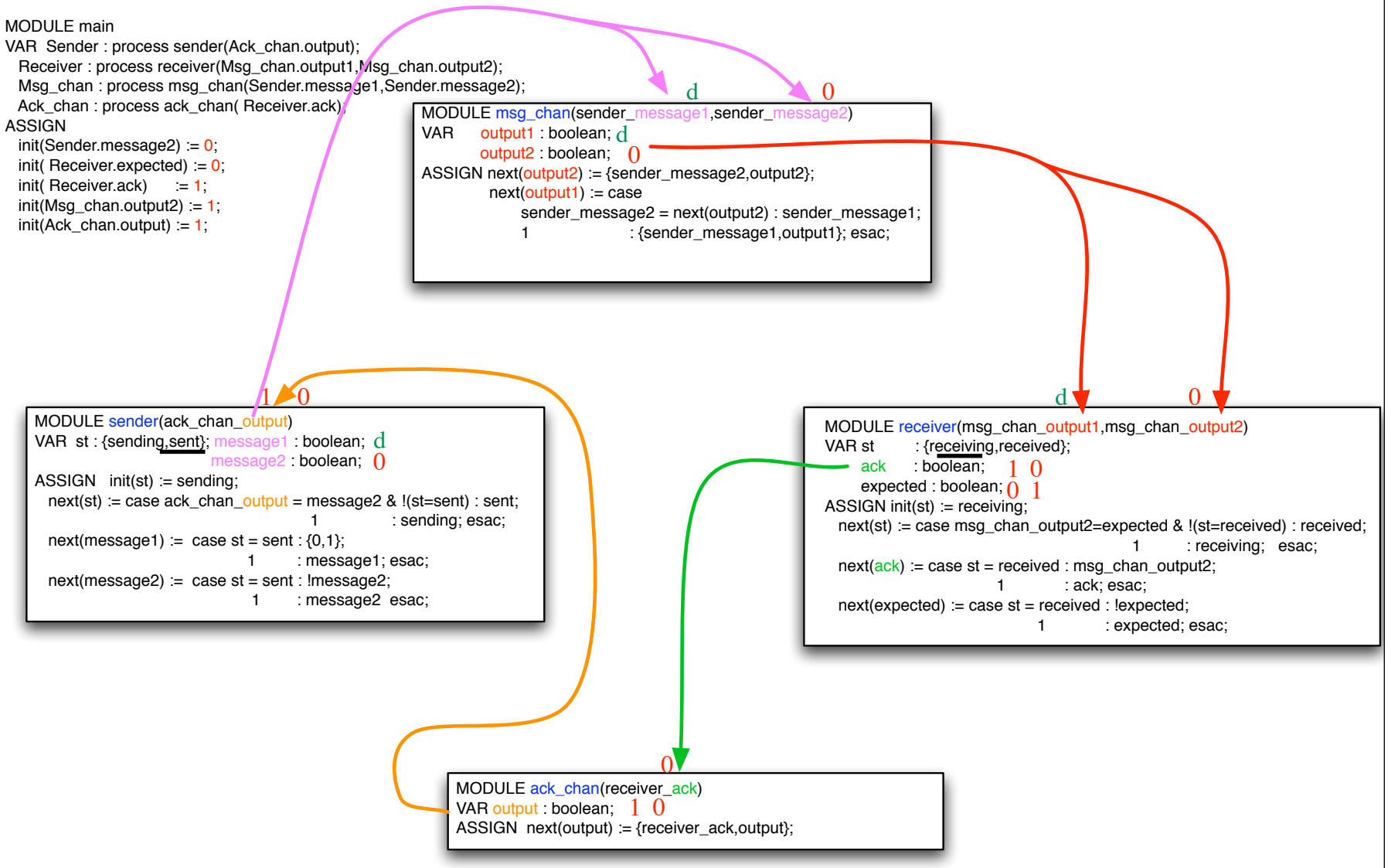

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1
expected : boolean; 0
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1
ASSIGN next(output) := {receiver_ack,output};

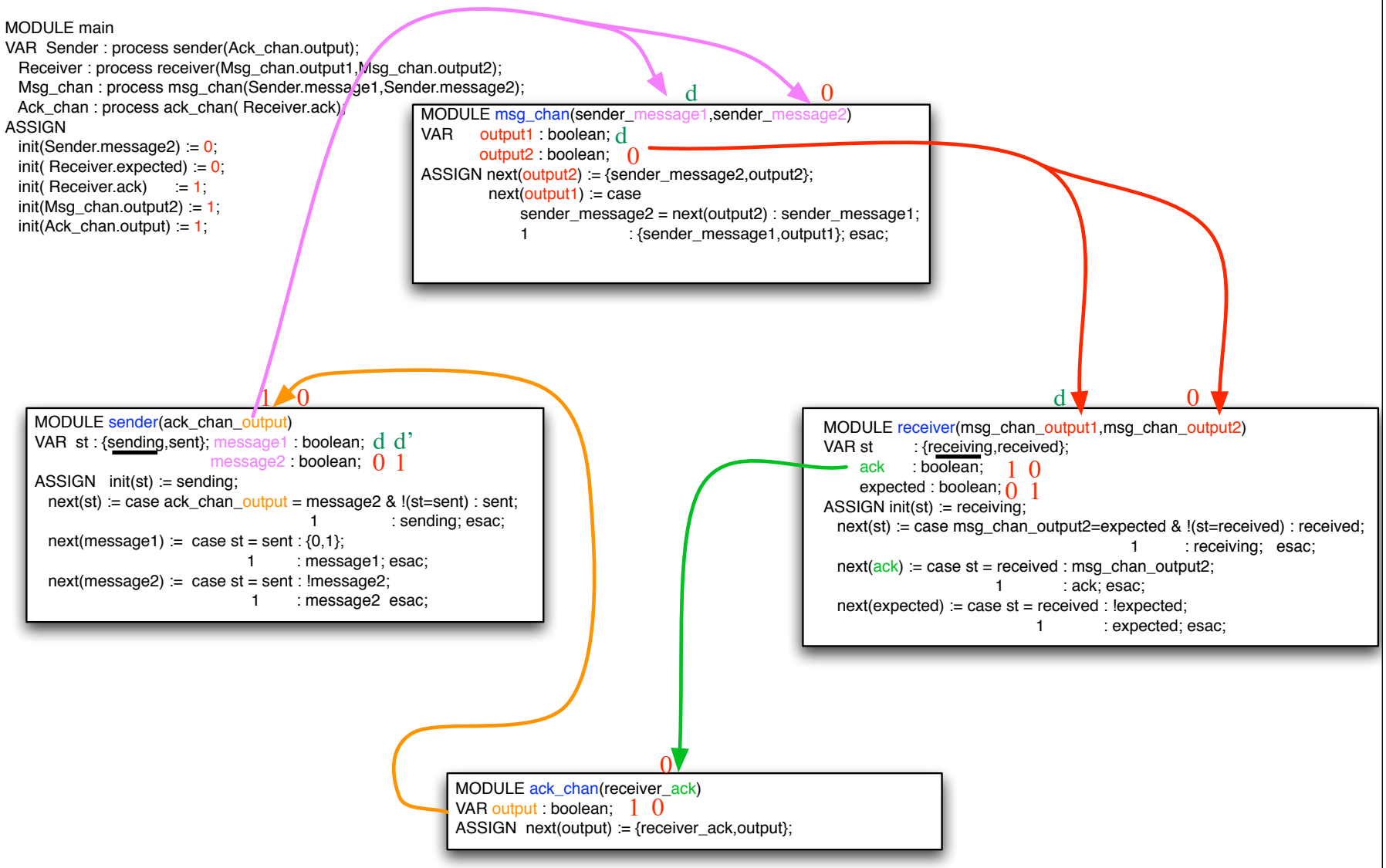

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1
ASSIGN next(output) := {receiver_ack,output};

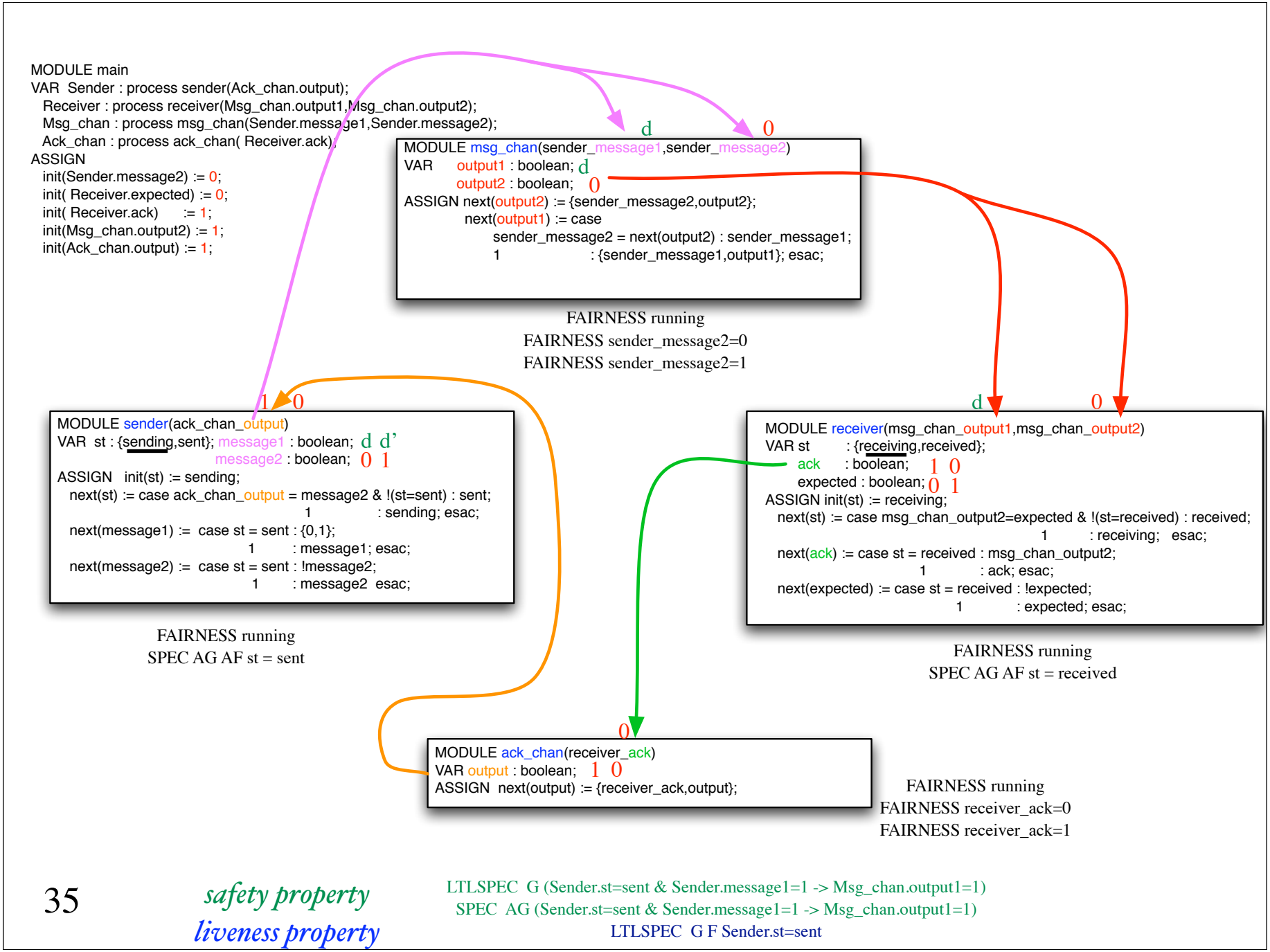

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1
ASSIGN next(output) := {receiver_ack,output};


MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
0
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1 0
ASSIGN next(output) := {receiver_ack,output};


MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1 0
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
0
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1 0
ASSIGN next(output) := {receiver_ack,output};


MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1 0
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d
message2 : boolean; 0
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
0
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1 0
ASSIGN next(output) := {receiver_ack,output};


MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
1 0
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d d'
message2 : boolean; 0 1
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
0
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1 0
ASSIGN next(output) := {receiver_ack,output};



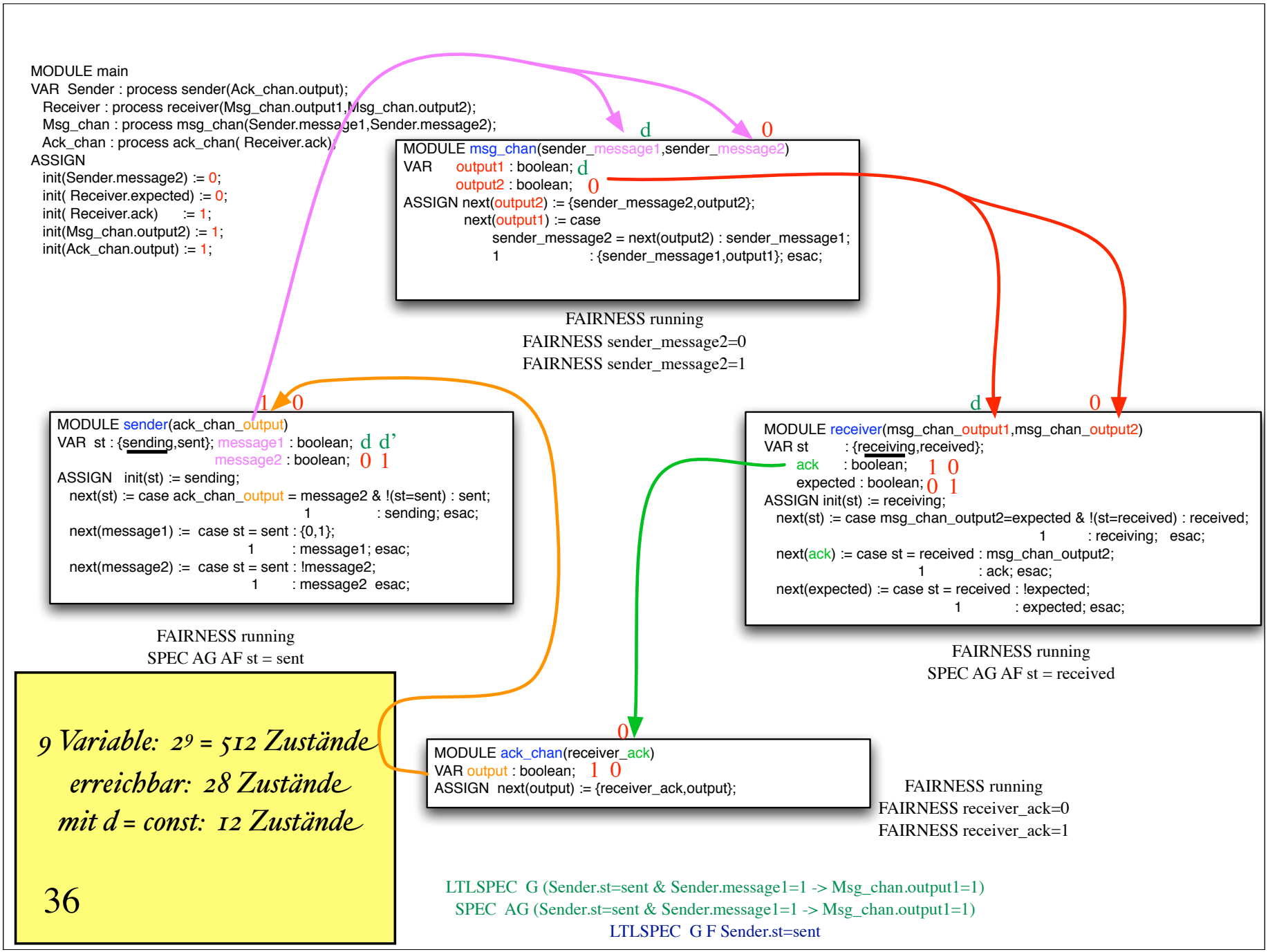
safety property

liveness property

LTLSPEC G (Sender.st=sent & Sender.message1=1 -> Msg_chan.output1=1)

SPEC AG (Sender.st=sent & Sender.message1=1 -> Msg_chan.output1=1)

LTLSPEC G F Sender.st=sent

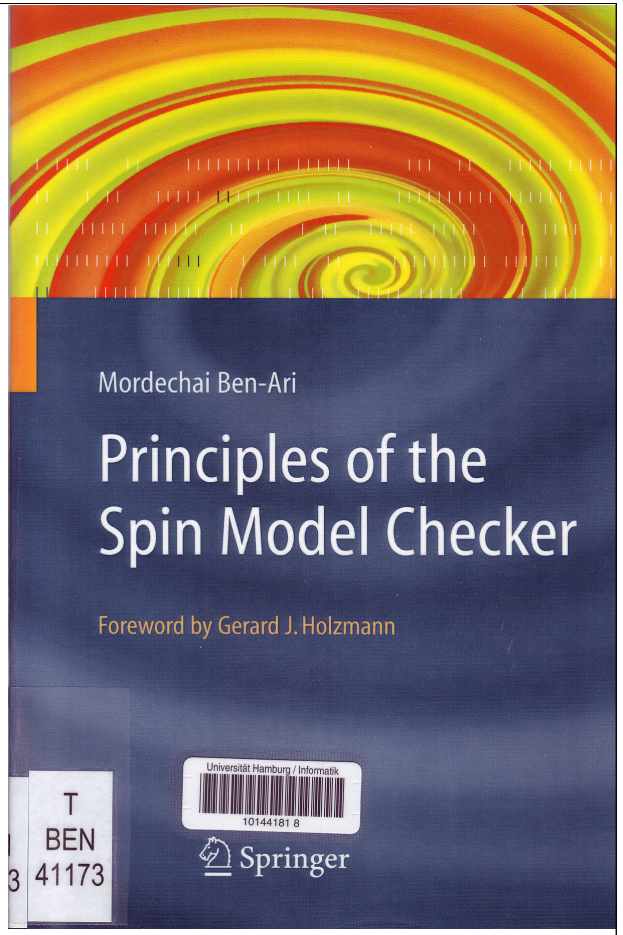

MODULE main
VAR Sender : process sender(Ack_chan.output);
Receiver : process receiver(Msg_chan.output1,Msg_chan.output2);
Msg_chan : process msg_chan(Sender.message1,Sender.message2);
Ack_chan : process ack_chan(Receiver.ack);
ASSIGN
init(Sender.message2) := 0;
init(Receiver.expected) := 0;
init(Receiver.ack) := 1;
init(Msg_chan.output2) := 1;
init(Ack_chan.output) := 1;
d
0
MODULE msg_chan(sender_message1,sender_message2)
VAR output1 : boolean; d
output2 : boolean; 0
ASSIGN next(output2) := {sender_message2,output2};
next(output1) := case
sender_message2 = next(output2) : sender_message1;
1 : {sender_message1,output1}; esac;
FAIRNESS running
FAIRNESS sender_message2=0
FAIRNESS sender_message2=1
1
0
MODULE sender(ack_chan_output)
VAR st : {sending,sent}; message1 : boolean; d d'
message2 : boolean; 0 1
ASSIGN init(st) := sending;
next(st) := case ack_chan_output = message2 & !(st=sent) : sent;
1 : sending; esac;
next(message1) := case st = sent : {0,1};
1 : message1; esac;
next(message2) := case st = sent : !message2;
1 : message2 esac;
FAIRNESS running
SPEC AG AF st = sent
d
0
MODULE receiver(msg_chan_output1,msg_chan_output2)
VAR st : {receiving,received};
ack : boolean; 1 0
expected : boolean; 0 1
ASSIGN init(st) := receiving;
next(st) := case msg_chan_output2=expected & !(st=received) : received;
1 : receiving; esac;
next(ack) := case st = received : msg_chan_output2;
1 : ack; esac;
next(expected) := case st = received : !expected;
1 : expected; esac;
FAIRNESS running
SPEC AG AF st = received
9 Variable: 2⁹ = 512 Zustände
erreichbar: 28 Zustände
mit d = const: 12 Zustände
36
0
MODULE ack_chan(receiver_ack)
VAR output : boolean; 1 0
ASSIGN next(output) := {receiver_ack,output};
FAIRNESS running
FAIRNESS receiver_ack=0
FAIRNESS receiver_ack=1
LTLSPEC G (Sender.st=sent & Sender.message1=1 -> Msg_chan.output1=1)
SPEC AG (Sender.st=sent & Sender.message1=1 -> Msg_chan.output1=1)
LTLSPEC G F Sender.st=sent

5.4.1 Expressing liveness properties in SPIN

Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

Listing 5.1. Critical section with starvation

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```



5.4.1 Expressing liveness properties in SPIN

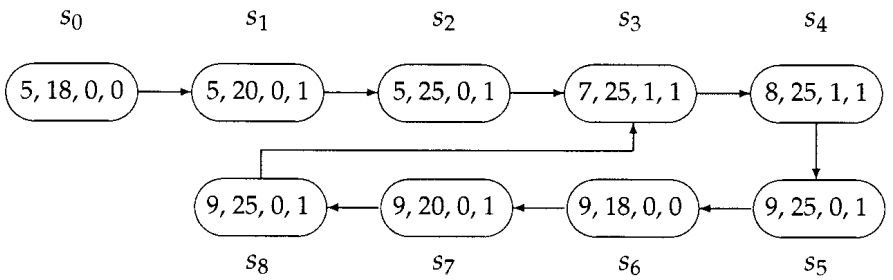
Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

Listing 5.1. Critical section with starvati

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```


$$
\begin{aligned}
s_0 &= \texttt{(5. wantP=1, 18. wantQ=1, 0, 0)} \longrightarrow \\
s_1 &= \texttt{(5. wantP=1, 20. wantP, 0, 1)} \longrightarrow \\
s_2 &= \texttt{(5. wantP=1, 25. wantQ=0, 0, 1)} \longrightarrow \\
s_3 &= \texttt{(7. wantQ, 25. wantQ=0, 1, 1)} \longrightarrow \\
s_4 &= \texttt{(8. wantP=0, 25. wantQ=0, 1, 1)} \longrightarrow \\
s_5 &= \texttt{(9. wantP=1, 25. wantQ=0, 0, 1)} \longrightarrow \\
s_6 &= \texttt{(9. wantP=1, 18. wantQ=1, 0, 0)} \longrightarrow \\
s_7 &= \texttt{(9. wantP=1, 20. wantP, 0, 1)} \longrightarrow \\
s_8 &= \texttt{(9. wantP=1, 25. wantQ=0, 0, 1)} \longrightarrow \\
s_9 &= \texttt{(7. wantQ, 25. wantQ=0, 1, 1)}
\end{aligned}
$$

Since state s_9 is the same as state s_3, they can be identified and the sequence of states extended to an infinite computation:



The critical section of process P (line 12) does not appear in any state of this computation, demonstrating that absence of starvation does not hold for this program.

5.4.1 Expressing liveness properties in SPIN

Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

Listing 5.1. Critical section with starvation

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```

5.4.2 Verifying liveness properties in SPIN

Add the statements

```
csp = true;
csp = false;
```

between lines 11 and 12 of the program in Listing 5.1; then the LTL formula <>csp expresses absence of starvation for process P. The verification of the temporal formula is carried out in a manner similar to that of the safety property, except that it must be performed in a mode called searching for *acceptance cycles* (Section 10.3.2). *Weak fairness*, explained in Section 5.5, must also be specified when this program is verified.

5.4.1 Expressing liveness properties in SPIN

Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

Listing 5.1. Critical section with starvation

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```

jSpin

Select Acceptance instead of Safety from the pulldown menu, and ensure that the box labeled Weak fairness is checked. Select Verify.

Command line

Run the verifier with the -a (*a*cceptance) argument and the -f (weak *f*airness) argument:[7]

```
spin -a -f "!<>csp" fourth-liveness.pml
gcc -o pan pan.c
pan -a -f
```

5.4.1 Expressing liveness properties in SPIN

Listing 5.1 shows a program for the critical section problem.[6] We leave it to the reader to verify that both mutual exclusion and absence of deadlock hold. Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

Listing 5.1. Critical section with starvation

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do
5    :: wantP = true;
6       do
7       :: wantQ ->
8            wantP = false;
9            wantP = true
10      :: else -> break
11      od;
12      wantP = false
13   od
14 }
15
16 active proctype Q() {
17   do
18   :: wantQ = true;
19      do
20      :: wantP ->
21           wantQ = false;
22           wantQ = true
23      :: else -> break
24      od;
25      wantQ = false
26   od
27 }
```

```
pan: acceptance cycle (at depth 14)
```

For safety properties, a counterexample consists of one state where the formula is false, but for a liveness property, a counterexample is an infinite computation in which something good – in this case, csp becomes true – never happens. To produce the counterexample, run a guided simulation with the trail. The output from JSPIN is:

```
Process Statement          wantQ      wantP
1 Q     wantQ = 1
1 Q     wantQ = 0          1
0 P     wantP = 1          0
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantQ              1          1
<<<<<START OF CYCLE>>>>>
1 Q     wantQ = 0          1          1
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantP = 0          1          1
1 Q     wantQ = 0          0          1
1 Q     wantQ = 1          0          0
0 P     wantP = 1          0          1
0 P     wantQ              1          1
1 Q     wantQ = 0          1          1
0 P     wantP = 0          1          0
1 Q     wantQ = 1          0          0
1 Q     wantQ = 0          0          1
0 P     wantP = 1          0          0
1 Q     wantQ = 1          1          0
1 Q     wantP              1          1
0 P     wantQ              1          1
spin: trail ends after 50 steps
```

The line START OF CYCLE indicates that the subsequent states form a cycle that can be repeated indefinitely. Since a variable appears in the SPIN output only when it is assigned to, the absence of a value for csp means that the variable has never been assigned to and hence that starvation occurs in this computation.