Synchronous composition of rewrite systems.

Narciso Martí-Oliet
Joint work with Óscar Martín
and Alberto Verdejo

Universidad Complutense de Madrid

Rio de Janeiro Oct 2018
Introductory example.
Train 1

station → to X → before X → crossing → after X → to station

MUTEX rem grant 1
grant 2

2/59
In modules TRAIN1 and TRAIN2:

```plaintext
ops atStation beforeX afterX : -> State .
ops toX crossing toStation : -> Trans .
rl atStation = [ toX ]=> beforeX .
rl beforeX = [ crossing ]=> afterX .
rl afterX = [ toStation ]=> atStation .
op isCrossing : -> Ppty{Bool} .
eq isCrossing @ crossing = true .
eq isCrossing @ G:Stage = false [owise] .
```

op rem : -> State .
op grant : Nat -> Trans .
rl rem = [ grant(I) ]=> rem .
op grants : Nat -> Ppty{Bool} .
eq grants(I) @ grant(I) = true .
eq grants(I) @ G:Stage = false [owise] .
sync TRAIN1 || MUTEX || TRAIN2 on MUTEX$grants(1) = TRAIN1$isCrossing /
MUTEX$grants(2) = TRAIN2$isCrossing .
```
In modules TRAIN1 and TRAIN2:

\[
\text{ops atStation beforeX afterX} : \to \text{State}.
\]
\[
\text{ops toX crossing toStation} : \to \text{Trans}.
\]
\[
\text{rl atStation} =\{ \text{toX}\} \Rightarrow \text{beforeX}.
\]
\[
\text{rl beforeX} =\{ \text{crossing}\} \Rightarrow \text{afterX}.
\]
\[
\text{rl afterX} =\{ \text{toStation}\} \Rightarrow \text{atStation}.
\]
\[
\text{op isCrossing} : \to \text{Ppty\{Bool\}}.
\]
\[
\text{eq isCrossing} @ \text{crossing} = \text{true}.
\]
\[
\text{eq isCrossing} @ G:\text{Stage} = \text{false} \text{ [owise]}.
\]

In module MUTEX:

\[
\text{op rem} : \to \text{State}.
\]
\[
\text{op grant} : \text{Nat} \to \text{Trans}.
\]
\[
\text{rl rem} =\{ \text{grant(I)}\} \Rightarrow \text{rem}.
\]
\[
\text{op grants} : \text{Nat} \to \text{Ppty\{Bool\}}.
\]
\[
\text{eq grants(I)} @ \text{grant(I)} = \text{true}.
\]
\[
\text{eq grants(I)} @ G:\text{Stage} = \text{false} \text{ [owise]}.
\]
In modules TRAIN1 and TRAIN2:

```plaintext
ops atStation beforeX afterX : -> State .
ops toX crossing toStation : -> Trans .
rl atStation = [ toX ]=> beforeX .
rl beforeX = [ crossing ]=> afterX .
rl afterX = [ toStation ]=> atStation .

op isCrossing : -> Ppty{Bool} .
eq isCrossing @ crossing = true .
eq isCrossing @ G:Stage = false [owise] .
```

In module MUTEX:

```plaintext
op rem : -> State .
op grant : Nat -> Trans .
rl rem = [ grant(I) ]=> rem .

op grants : Nat -> Ppty{Bool} .
eq grants(I) @ grant(I) = true .
eq grants(I) @ G:Stage = false [owise] .
```

```
sync TRAIN1 || MUTEX || TRAIN2
  on MUTEX$grants(1) = TRAIN1$isCrossing \
  \ MUTEX$grants(2) = TRAIN2$isCrossing .
```
Why existing tools are not enough.
Synchronous communication:

```
rl   < rem >   < Train1 | beforeX >
=> < grant(1) > < Train1 | crossing > .
```
Synchronous communication:

\[ r1 \ < \ rem > \ < \ Train1 \ | \ beforeX > \]
\[ \rightarrow \ < \ grant(1) > \ < \ Train1 \ | \ crossing > \].

⚠️ No modularity.
Asynchronous communication by message passing:

rl  < rem >
    => < grant(1) > < msg grant Train1 > .

rl  < Train1 | beforeX > < msg grant Train1 >
    => < Train1 | crossing > .
Asynchronous communication by message passing:

```
rl  < rem >
    => < grant(1) > < msg grant Train1 > .

rl  < Train1 | beforeX > < msg grant Train1 >
    => < Train1 | crossing > .
```

✓ Modularity, but...
Asynchronous communication by message passing:

\[
\text{rl} \quad < \text{rem} > \\
=> < \text{grant}(1) > < \text{msg grant Train1} > .
\]

\[
\text{rl} \quad < \text{Train1 | beforeX} > < \text{msg grant Train1} > \\
=> < \text{Train1 | crossing} > .
\]

✓ Modularity, but...

⚠️ asynchrony does not always fit, specially for control
Asynchronous communication by message passing:

\[
\text{rl} \ < \text{rem} > \\
\Rightarrow < \text{grant(1)} > < \text{msg grant Train1} > .
\]

\[
\text{rl} \ < \text{Train1 | beforeX} > < \text{msg grant Train1} > \\
\Rightarrow < \text{Train1 | crossing} > .
\]

✓ Modularity, but...

⚠ special cases: ordered messages, limited capacity channel
Asynchronous communication by message passing:

\[ \text{rl} \quad < \text{rem} > \]
\[ \Rightarrow < \text{grant}(1) > < \text{msg grant Train1} > . \]

\[ \text{rl} \quad < \text{Train1 | beforeX} > < \text{msg grant Train1} > \]
\[ \Rightarrow < \text{Train1 | crossing} > . \]

✓ Modularity, but... 

⚠️ strategic control
Let’s be egalitarian.
\textbf{MUTEX}

\begin{align*}
\text{op rem : } & \rightarrow \text{ State} . \\
\text{op grant : } & \text{Nat } \rightarrow \text{ Trans} . \\
\text{rl rem =}[ & \text{ grant(I) } ]\rightarrow \text{ rem} . \\
\text{op grants : } & \text{Nat } \rightarrow \text{ Ppty\{Bool\}} . \\
\text{eq grants(I) } & @ \text{ grant(I) } = \text{ true} . \\
\text{eq grants(I) } & @ \text{ G:Stage } = \text{ false [otherwise]} .
\end{align*}
For fairness conditions

to X → before X

to station → after X

other things

crossing
difficult route

station
For fairness conditions

- Station to X before X
- Crossing after X
- Difficult route
- Other things

wanting → crossing
For syncing

Train 1

station

to station

before X

to X

crossing

after X

MUTEX

grant 1

rem

grant 2
For identifying equal actions

writing 1

1 stored  2 stored  3 stored  ...

1 stored
fmod STAGE is
  sorts State Trans Stage .
  subsorts State Trans < Stage .
  op init : -> Stage .
endfm

aemod MUTEX is
  ex STAGE .
  ...
  op rem : -> State .
  op grant : Nat -> Trans .
  rl rem = [ grant(I) ] => rem .
  op grants : Nat -> Ppty{Bool} .
  eq grants(I) @ grant(I) = true .
  eq grants(I) @ G:Stage = false [owise] .
endaem
Properties
\texttt{TRAIN1}\$isCrossing = \texttt{MUTEX}\$grants(1)
fmod PPTY{X :: TRIV} is
  ex STAGE .
  sort Ppty{X} .
  op _@_ : Ppty{X} Stage ~> X$Elt .
endfm

fth TRIV is
  sort Elt .
endfth
aemod MUTEX is
  ex STAGE .
  ex PPTY{Bool} .
  op rem : -> State .
  op grant : Nat -> Trans .
  rl rem = [ grant(I) ] => rem .
  op grants : Nat -> Ppty{Bool} .
  eq grants(I) @ grant(I) = true .
  eq grants(I) @ G:Stage = false [owise] .
endaem

emod SAFE-TRAINS is
  sync TRAIN1 || MUTEX || TRAIN2
    on MUTEX$grants(1) = TRAIN1$isCrossing
       \ MUTEX$grants(2) = TRAIN2$isCrossing .
  ex PPTY{Bool} .
  var G : Stage .
  op isSomeCrossing : -> Ppty{Bool} .
  eq isSomeCrossing @ G = isCrossing @ TRAIN1(G)
    or isCrossing @ TRAIN2(G) .
endaem
aemod A is
  ex STAGE .
  op o : -> Stage .
  pr NAT .  ex PPTY{Nat} .
  op value : -> Ppty{Nat} .
  eq value @ o = 3 .
endaem

aemod B is
  ex STAGE .  pr NAT .
  op noValue : -> State .
  op value_ : Nat -> State .
  ops reading forgetting : -> Trans .
  var N : Nat .
  rl noValue = [ reading ]=> value N .
  rl value N = [ forgetting ]=> noValue .
  ex PPTY{Nat} .
  op value : -> Ppty{Nat} .
  eq value @ value N = N .
endaem

sync A || B
  on A$value = B$value .
The split.
split : egalitarian rewrite systems $\rightarrow$ standard rewrite systems

split : egalitarian transition structures $\rightarrow$ Kripke structures
\[ \text{rl} \text{ rem} = [\text{grant(I)}] => \text{rem} . \]

\[ \text{split} \]

\[ \text{rl rem} => \text{grant(I)} . \]
\[ \text{rl grant(I)} => \text{rem} . \]
\[ rl \text{ rem } = [ \text{ grant(I) } ] \Rightarrow \text{ rem} . \]

\[ \text{split} \]

\[ rl \text{ rem } \Rightarrow \text{ grant(I)} . \]

\[ rl \text{ grant(I) } \Rightarrow \text{ rem} . \]

\[ rl \text{ beforeX } \Rightarrow \text{ crossing} . \]

\[ \text{crl} < \text{ beforeX}, \text{ rem}, G > \Rightarrow < \text{ crossing}, \text{ grant(I)}, G > \]

\[ \text{if grants}(1) \odot \text{ grant(I)} = \text{ isCrossing} \odot \text{ crossing} \]

\[ \land \text{ grants}(2) \odot \text{ grant(I)} = \text{ isCrossing} \odot G . \]
split

\[
\text{atEgRwSys} \rightarrow \text{EgRwSys} \rightarrow \text{EgRwSys} \\
\text{atEgTrStr} \rightarrow \text{EgTrStr} \rightarrow \text{EgTrStr}
\]
Insertion sort.
Terminator

Insertion sort controller

Passive to swapping layer

Passive array
\textbf{aemod} ARRAY \textbf{is}
\begin{verbatim}
  ex STAGE .
  ex BOOL+NAT+PPTIES .
  sorts Cell CellSet .
  op (_,_) : Nat Nat -> Cell .  \textit{--- \textit{(index, contents)}}
  subsort Cell < CellSet .
  op __ : CellSet CellSet -> CellSet [\textit{comm assoc}] .
  op \{\_\} : CellSet -> State .
  op updating(_,_,_)\|_ : Nat Nat CellSet -> Trans .

  vars I V V' J W : Nat .
  var CS : CellSet .
  rl \{ (I, V) CS \}
    \=[ updating(I, V') | CS ]=>
     \{ (I, V') CS \} .

  op contents : Nat -> Ppty\{Nat\} .
  op isUpdating : \rightarrow Ppty\{Bool\} .
  eq contents(I) \@ \{ (I, V) CS \} = V .
  eq contents(I) \@ updating(I, V) \| CS = V .
  eq contents(I) \@ updating(J, W) \| (I, V) CS = V .
  eq isUpdating \@ updating(I, V) \| CS = \text{true} .
  eq isUpdating \@ \{ CS \} = \text{false} \[\text{owise}\] .
\end{verbatim}
endaem
ready to write
V to I

writing
V to I

written

... (I, 1) ...

... updating(I, V) | ...

... (I, V) ...

...
aemod SWAPPING-LAYER is
  ex MODES .
  ex BOOL+NAT+PPTIES .
op _,_,_,_,_,_ : Maybe{Nat} Maybe{Nat}
       Maybe{Bool} Maybe{Nat} Maybe{Nat} -> Data .
ops idle reqRecved dataRead halfWritten: -> StateMode .
ops recvingReq reading resetting writing : -> TransMode .

vars I1 I2 C1 C2 : Nat .
var S : Bool .
rl idle | nothing, nothing, nothing, nothing, nothing, nothing
  = reqRecved | I1, I2, S, nothing, nothing, nothing .
  rl reqRecved | I1, I2, S, nothing, nothing
     = reading | I1, I2, S, nothing, nothing ]=>
       dataRead | I1, I2, S, C1, C2 .
  rl dataRead | I1, I2, false, C1, C2
     = resetting | nothing, nothing, nothing, nothing, nothing, nothing ]=>
       idle | nothing, nothing, nothing, nothing, nothing, nothing .
  rl dataRead | I1, I2, true, C1, C2
     = writing | I1, I2, true, C1, C2 ]=>
       halfWritten | nothing, I2, nothing, C1, nothing .
  rl halfWritten | nothing, I2, true, C1, nothing
     = writing | nothing, I2, true, C1, nothing ]=>
       idle | nothing, nothing, nothing, nothing, nothing, nothing .
op contents : Nat -> Ppty{Nat}  .
op doUpdate : -> Ppty{Bool}  .
var D : Data  .
var G : Stage  .
eq doUpdate @ (writing | D) = true  .
eq doUpdate @ G = false [owise]  .
eq contents(I1) @ (writing | I1, I2, true, C1, C2) = C2  .
eq contents(I2) @ (writing | nothing, I2, true, C1, nothing) = C1  .
eq contents(I1) @ (dataRead | I1, I2, S, C1, C2) = C1  .
eq contents(I2) @ (dataRead | I1, I2, S, C1, C2) = C2  .

ops index1 index2 : -> Ppty{Maybe{Nat}}  .
ops swapOrComp is1>2 hasEnded : -> Ppty{Maybe{Bool}}  .
eq index1 @ (reqRecved | I1, I2, S, nothing, nothing) = I1  .
eq index2 @ (reqRecved | I1, I2, S, nothing, nothing) = I2  .
eq swapOrComp @ (reqRecved | I1, I2, S, nothing, nothing) = S  .
eq index1 @ G = nothing [owise]  .
eq index2 @ G = nothing [owise]  .
eq swapOrComp @ G = nothing [owise]  .
eq is1>2 @ (dataRead | I1, I2, false, C1, C2) = (C1 > C2)  .
eq is1>2 @ G = nothing [owise]  .
eq hasEnded @ (idle | D) = true  .
eq hasEnded @ G = false [owise]  .
endaem
aemod INSERTION-SORT is
  ex MODES .
  ex BOOL+NAT+PPTIES .
  ops readyToMove justMoved : -> StateMode .
  ops mainGoingRight secGoingLeft seeingIfReached : -> TransMode .
  op _,_,_,_ : Nat Nat Maybe{Bool} : -> Data .

vars I J : Nat .
var R : Bool .
rl  readyToMove | I, s J, false
     = [ secGoingLeft | I, J, nothing ] =>
         justMoved | I, J, nothing .  --- when not reached

rl  readyToMove | I, J, true
     = [ mainGoingRight | s I, s I, nothing ] =>
         justMoved | s I, s I, nothing .  --- when reached

rl  justMoved | I, s J, nothing
     = [ seeingIfReached | I, s J, nothing ] =>
         readyToMove | I, s J, R .

rl  justMoved | I, 0, nothing
     = [ seeingIfReached | I, 0, nothing ] =>
         readyToMove | I, 0, true .  --- at array head
ops index1 index2 : -> Ppty{Maybe{Nat}} .
ops swapOrComp is1>2 : -> Ppty{Maybe{Bool}} .
var G : Stage .

eq index1 @ (secGoingLeft | I, J, nothing) = J .
eq index2 @ (secGoingLeft | I, J, nothing) = s J .
eq swapOrComp @ (secGoingLeft | I, J, nothing) = true . --- swap

eq index1 @ (seeingIfReached | I, s J, nothing) = J .
eq index2 @ (seeingIfReached | I, s J, nothing) = s J .
eq swapOrComp @ (seeingIfReached | I, s J, nothing) = false . --- comp

eq index1 @ G = nothing [owise] .
eq index2 @ G = nothing [owise] .
eq swapOrComp @ G = nothing [owise] .
eq is1>2 @ (readyToMove | I, s J, R) = R .
eq is1>2 @ G = nothing [owise] .
endaem
emod ARRAY-SORTING is
  sync INSERTION-SORT || SWAPPING-LAYER || ARRAY
  on INSERTION-SORT$index1 = SWAPPING-LAYER$index1
    \ INSERTION-SORT$index2 = SWAPPING-LAYER$index2
    \ INSERTION-SORT$swapOrComp = SWAPPING-LAYER$swapOrComp
    \ INSERTION-SORT$is1>2 = SWAPPING-LAYER$is1>2
    \ SWAPPING-LAYER$contents = ARRAY$contents
    \ SWAPPING-LAYER$doUpdate = ARRAY$isUpdating .
endem
Parameterized programming.
emod ARRAY-SORTING-BPRINT{S :: SORTER-IFACE,
   L :: SWAPPING-LAYER-IFACE,
   A :: ARRAY-IFACE} is
	sync S || L || A
   on S$index1 = L$index1
      \ S$index2 = L$index2
      \ S$swapOrComp = L$swapOrComp
      \ S$is1>2 = L$is1>2
      \ S$hasEnded = L$hasEnded
      \ L$contents = A$contents
      \ L$doUpdate = A$isUpdating .
endem
th ARRAY-IFACE is
  ex BOOL+NAT+PPTIES .
  op contents : Nat -> Ppty{Nat} .
  op isUpdating : -> Ppty{Bool} .
endth

th SWAPPING-LAYER-IFACE is
  ex BOOL+NAT+PPTIES .
  op contents : Nat -> Ppty{Nat} .
  op doUpdate : -> Ppty{Bool} .
  ops index1 index2 : -> Ppty{Maybe{Nat}} .
  ops swapOrComp is1>2 hasEnded : -> Ppty{Maybe{Bool}} .
endth

th SORTER-IFACE is
  ex BOOL+NAT+PPTIES .
  ops index1 index2 : -> Ppty{Maybe{Nat}} .
  ops swapOrComp is1>2 hasEnded : -> Ppty{Maybe{Bool}} .
endth
view Array
   from ARRAY-IFACE to ARRAY is
   op contents to contents .
   op isUpdating to isUpdating .
endv

view SwappingLayer
   from SWAPPING-LAYER-IFACE to SWAPPING-LAYER is
endv
view Array
  from ARRAY-IFACE to ARRAY is
  op contents to contents .
  op isUpdating to isUpdating .
endv

view SwappingLayer
  from SWAPPING-LAYER-IFACE to SWAPPING-LAYER is
endv

view InsertionSorter
  from SORTER-IFACE to INSERTION-SORT is
  ...
  op hasEnded to ??? .
endv
aemod NAT-TERMINATOR is
  ex STAGE .
  ex BOOL+NAT+PPTIES .
  op dummy : -> State .
  op undefined : -> Ppty{Nat} .
endaem
aemod NAT-TERMINATOR is
  ex STAGE .
  ex BOOL+NAT+PPTIES .
  op dummy : -> State .
  op undefd : -> Ppty{Nat} .
endaem

emod INSERTION-SORT+NAT-TERMINATOR is
  sync INSERTION-SORT || NAT-TERMINATOR
    on no-criteria .
  op index1 : -> Ppty{Nat} .
  eq index1 @ G = index1 @ INSERTION-SORT(G) .
  ...
  op hasEnded : -> Ppty{Bool} .
  eq hasEnded @ G = undefd @ NAT-TERMINATOR(G) .
endem

view InsertionSorter
  from SORTER-IFACE to INSERTION-SORT+NAT-TERMINATOR is
denv
aemod NAT-TERMINATOR is
   ex STAGE .
   ex BOOL+NAT+PPTIES .
   op dummy : -> State .
   op undefined : -> Ppty{Nat} .
endaem

emod INSERTION-SORT+NAT-TERMINATOR is
   sync INSERTION-SORT || NAT-TERMINATOR
      on no-criteria .
   op index1 : -> Ppty{Nat} .
   eq index1 @ G = index1 @ INSERTION-SORT(G) .
   ... 
   op hasEnded : -> Ppty{Bool} .
   eq hasEnded @ G = undefined @ NAT-TERMINATOR(G) .
endaem

view InsertionSorter
   from SORTER-IFACE to INSERTION-SORT+NAT-TERMINATOR is
enda

emod ARRAY-SORTING is
   pr ARRAY-SORTING-BPRINT
      {InsertionSorter, SwappingLayer, Array} .
endaem
Alternating bit protocol.
Interface for producer and consumer:

```plaintext
th PROCESS-IFACE{Msg :: TRIV} is
  ex PPTY{Maybe{Msg}} .
  op msgMoving : -> Ppty{Maybe{Msg}} .
endth
```
Interface for producer and consumer:

```plaintext
th PROCESS-IFACE{Msg :: TRIV} is
  ex PPTY{Maybe{Msg}} .
  op msgMoving : -> Ppty{Maybe{Msg}} .
endth
```

Interface for sender and receiver:

```plaintext
th PROTOCOL-IFACE{ProcMsg :: TRIV,
  Pck2Chnl :: TRIV,
  PckFChnl :: TRIV} is
  ex PPTY{Maybe{ProcMsg}} .
  ex PPTY{Maybe{Pck2Chnl}} .
  ex PPTY{Maybe{PckFChnl}} .
  op procMsgMoving : -> Ppty{Maybe{ProcMsg}} .
  op pckLeaving2Chnl : -> Ppty{Maybe{Pck2Chnl}} .
  op pckComingFChnl : -> Ppty{Maybe{PckFChnl}} .
endth
```
Interface for producer and consumer:

\[
\text{th \ PROCESS-IFACE\{Msg :: TRIV\} is}
\]
\[
\text{ex PPTY\{Maybe\{Msg\}\} .}
\]
\[
\text{op msgMoving : \rightarrow Ppty\{Maybe\{Msg\}\} .}
\]
\[
\text{endth}
\]

Interface for sender and receiver:

\[
\text{th \ PROTOCOL-IFACE\{ProcMsg :: TRIV, Pck2Chnl :: TRIV, PckFChnl :: TRIV\} is}
\]
\[
\text{ex PPTY\{Maybe\{ProcMsg\}\} .}
\]
\[
\text{ex PPTY\{Maybe\{Pck2Chnl\}\} .}
\]
\[
\text{ex PPTY\{Maybe\{PckFChnl\}\} .}
\]
\[
\text{op procMsgMoving : \rightarrow Ppty\{Maybe\{ProcMsg\}\} .}
\]
\[
\text{op pckLeaving2Chnl : \rightarrow Ppty\{Maybe\{Pck2Chnl\}\} .}
\]
\[
\text{op pckComingFChnl : \rightarrow Ppty\{Maybe\{PckFChnl\}\} .}
\]
\[
\text{endth}
\]

Interface for channels:

\[
\text{th \ CHANNEL-IFACE\{Pck :: TRIV\} is}
\]
\[
\text{ex PPTY\{Maybe\{Pck\}\} .}
\]
\[
\text{ops pckComing pckLeaving : \rightarrow Ppty\{Maybe\{Pck\}\} .}
\]
\[
\text{endth}
\]
\textbf{emod} \textbf{COMM-SYSTEM-BPRINT}

\begin{verbatim}
{ Sndr :: PROTOCOL-IFACE{{Msg :: TRIV,
    MsgPck :: TRIV,
    AckPck :: TRIV},
    MsgChnl :: CHANNEL-IFACE{MsgPck :: TRIV},
    AckChnl :: CHANNEL-IFACE{AckPck :: TRIV},
    Rcvr :: PROTOCOL-IFACE{{Msg :: TRIV,
    AckPck :: TRIV,
    MsgPck :: TRIV}}}

} is
sync Sndr || MsgChnl || AckChnl || Rcvr
   on Sndr$pckLeaving2Chnl = MsgChnl$pckComing
    \ /\ MsgChnl$pckLeaving = Rcvr$pckComingFChnl
    \ /\ Rcvr$pckLeaving2Chnl = AckChnl$pckComing
    \ /\ AckChnl$pckLeaving = Sndr$pckComingFChnl .

\end{verbatim}
fmod PACKET-BUILDER{Cnt :: TRIV, Wrp :: TRIV} is
    sort Packet{Cnt, Wrp} .
    op packet : Cnt$Elt Wrp$Elt -> Packet{Cnt, Wrp} .
endfm

view Packet{Cnt :: TRIV, Wrp :: TRIV}
    from TRIV to PACKET-BUILDER{Cnt, Wrp} is
    sort Elt to Packet{Cnt, Wrp} .
endv
fmod ACK is
    sort Ack .
    op ack : -> Ack .
endfm

view Ack
    from TRIV to ACK is
    sort Elt to Ack .
endv
aemod ABP-SENDER\{Msg :: TRIV\} is

ex MODES .

pr MAYBE\{Msg\} .  pr MAYBE\{Bool\} .

ops ready2Send waiting4Ack ready4Msg : -> StateMode .

ops takingMsg sending receivingAck : -> TransMode .

op _,_,_,_ : Maybe\{Msg\} Bool Maybe\{Bool\} -> Data .

var M : Msg$Elt .  var B : Bool .  var D : Data .

rl ready4Msg | nothing, B, nothing

  = [ takingMsg | nothing, B, nothing ]=>

    ready2Send | M, B, nothing .

rl ready2Send | D

  = [ sending | D ]=>

    waiting4Ack | D .

rl waiting4Ack | D

  = [ sending | D ]=>

    waiting4Ack | D .

rl waiting4Ack | M, B, nothing

  = [ receivingAck | M, B, not B ]=>

    waiting4Ack | M, B, nothing .

rl waiting4Ack | M, B, nothing

  = [ receivingAck | M, B, B ]=>

    ready4Msg | nothing, not B, nothing .
pr PACKET-BUILDER{Msg, Bool}  
pr PACKET-BUILDER{Ack, Bool}  
ex PPTY{Maybe{Msg}}  
ex PPTY{Maybe{Packet{Msg, Bool}}}  
ex PPTY{Maybe{Packet{Ack, Bool}}}  
op procMsgMoving : -> Ppty{Maybe{Msg}}  
op pckLeaving2Chnl : -> Ppty{Maybe{Packet{Msg, Bool}}}  
op pckComingFChnl : -> Ppty{Maybe{Packet{Ack, Bool}}}  
var G : Stage  
var M : Msg$Elt  
vars B B' : Bool  
eq procMsgMoving @ (takingMsg | M, B, nothing) = M  
eq procMsgMoving @ G = nothing [owise]  
eq pckLeaving2Chnl @ (sending | M, B, nothing) = packet(M, B)  
eq pckLeaving2Chnl @ G = nothing [owise]  
eq pckComingFChnl @ (receivingAck | M, B, B') = packet(ack, B')  
eq pckComingFChnl @ G = nothing [owise]  
endaem
view AbpSender{Msg :: TRIV}
  from PROTOCOL-IFACE{Msg,
    Packet{Msg, Bool},
    Packet{Ack, Bool}}
  to ABP-SENDER{Msg} is
  op procMsgMoving to procMsgMoving .
  op pckLeaving2Chnl to pckLeaving2Chnl .
  op pckComingFChnl to pckComingFChnl .
endv
aemod ABP-RECEIVER{Msg :: TRIV} is

  ex MODES .
  pr MAYBE{Msg} . pr MAYBE{Bool} .
  ops ready2SendAck waiting4Pck pckReceived : -> StateMode .
  ops givingMsg sendingAck receivingPck : -> TransMode .
  op _,_,_,_ : Maybe{Msg} Bool Maybe{Bool} -> Data .
  var M : Msg$Elt .
  var B : Bool .
  var D : Data .

  rl pckReceived | M, B, nothing
                  = [ givingMsg | M, B, nothing ] =>
                 ready2SendAck | nothing, B, nothing .

  rl ready2SendAck | D
                   = [ sendingAck | D ] =>
                 waiting4Pck | D .

  rl waiting4Pck | D
                   = [ sendingAck | D ] =>
                 waiting4Pck | D .

  rl waiting4Pck | nothing, B, nothing
                   = [ receivingPck | M, B, B ] =>
                 waiting4Pck | nothing, B, nothing .

  rl waiting4Pck | nothing, B, nothing
                   = [ receivingPck | M, B, not B ] =>
                 pckReceived | M, not B, nothing .
pr PACKET-BUILDER{Msg, Bool}.
pr PACKET-BUILDER{Ack, Bool}.
ex PPTY{Maybe{Msg}}.
ex PPTY{Maybe{Packet{Msg, Bool}}}.
ex PPTY{Maybe{Packet{Ack, Bool}}}.
op procMsgMoving : -> Ppty{Maybe{Msg}}.
op pckLeaving2Chnl : -> Ppty{Maybe{Packet{Ack, Bool}}}.
op pckComingFChnl : -> Ppty{Maybe{Packet{Msg, Bool}}}.
var G : Stage.
var M : Msg$Elt.
vars B B' : Bool.
eq procMsgMoving @ (givingMsg | M, B, nothing) = M.
eq procMsgMoving @ G = nothing [owise].
eq pckLeaving2Chnl @ (sendingAck | nothing, B, nothing) = packet(ack, B).
eq pckLeaving2Chnl @ G = nothing [owise].
eq pckComingFChnl @ (receivingPck | M, B, not B) = packet(M, B).
eq pckComingFChnl @ G = nothing [owise].
endaem
view AbpReceiver{Msg :: TRIV}  
from PROTOCOL-IFACE{Msg,  
    Packet{Msg, Bool},  
    Packet{Ack, Bool}}  
  to ABP-RECEIVER{Msg} is  
op procMsgMoving to procMsgMoving .  
op pckLeaving2Chnl to pckLeaving2Chnl .  
op pckComingFChnl to pckComingFChnl .  
endv
aemod CHANNEL{Pck :: TRIV} is
  ex STAGES .
  pr MAYBE{Pck} .
  subsort Maybe{Pck} < State .
  ops acceptingPck deliveringPck : Pck$Elt -> Trans .
  op losingPck : -> Trans .
  var P : Pck$Elt .
  rl nothing =\[ acceptingPck(P) \]=> P .
  rl P =\[ deliveringPck(P) \]=> nothing .
  rl P =\[ losingPck \]=> nothing .
ex PPTY{Maybe{Pck}} .
op pckComing pckLeaving : -> Ppty{Maybe{Pck}} .
var P : Pck$Elt .
var G : Stage .
eq pckComing @ acceptingPck(P) = P .
eq pckComing @ G = nothing [owise] .
eq pckLeaving @ deliveringPck(P) = P .
eq pckLeaving @ G = nothing [owise] .
endaem
view Channel{Pck :: TRIV}
    from CHANNEL-IFACE{Pck}
    to CHANNEL{Pck} is
    op pckComing to pckComing .
    op pckLeaving to pckLeaving .
endv
The final system specification:

\[
\text{emod } \text{ABP-SYSTEM}\{\text{Msg} :: \text{TRIV}\} \text{ is } \\
\text{pr } \text{COMM-SYSTEM-BPRINT}\{\text{AbpSender}\{\text{Msg}\},} \\
\text{Channel}\{\text{Packet}\{\text{Msg}, \text{Bool}\}\},} \\
\text{Channel}\{\text{Packet}\{\text{Ack}, \text{Bool}\}\},} \\
\text{AbpReceiver}\{\text{Msg}\}\} . \\
\text{endem}
\]
That system can be viewed as a channel:

```plaintext
e mod COMM-SYSTEM-BPRINT-PPT
   { Sndr :: PROTOCOL-IFACE{Msg :: TRIV,
       MsgPacket :: TRIV,
       AckPacket :: TRIV},
    MsgChnl :: CHANNEL-IFACE{MsgPacket :: TRIV},
    AckChnl :: CHANNEL-IFACE{AckPacket :: TRIV},
    Rcvr :: PROTOCOL-IFACE{Msg :: TRIV,
       AckPacket :: TRIV,
       MsgPacket :: TRIV}
} is
  pr COMM-SYSTEM-BPRINT{Sndr, MsgChnl, AckChnl, Rcvr} .
  ex PPTY{Maybe{Msg}} .
  ops msgComing msgLeaving : -> Ppty{Maybe{Msg}} .
  var G : Stage .
  eq msgComing @ G = procMsgMoving @ Sndr(G) .
  eq msgLeaving @ G = procMsgMoving @ Rcvr(G) .
endem
```
view CommSystemAsChannel\{Msg :: TRIV\}
from CHANNEL-IFACE\{Msg\}
to COMM-SYSTEM-BPRINT-PPT\{AbpSender\{Msg\},
                         Channel\{Packet\{Msg, Bool\}\},
                         Channel\{Packet\{Ack, Bool\}\},
                         AbpReceiver\{Msg\}\}
   is
     op pckComing to msgComing .
     op pckLeaving to msgLeaving .
endv
References


- *Alternating bit protocol as an example of compositional system specification*, Ó. Martín, A. Verdejo, N. Martí-Oliet (technical report).

- http://maude.sip.ucm.es/syncprod
Synchronous composition of rewrite systems

`sync YOU || I`

`on YOU$have_listened = I$thank_you`

`/\ YOU$ask_questions = I$will_try_to_answer`