Alternating bit protocol
as an example of
compositional system specification*

Technical Report 01/18

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet
{omartins,jalberto,narciso}@ucm.es

Departamento de Sistemas Informáticos y Computación
Facultad de Informática, Universidad Complutense de Madrid, Spain

Jan 2018

*Partially supported by MINECO Spanish project TRACES (TIN2015-67522-C3-3-R),
and Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731).
Abstract

We show a complete modular specification of the alternating bit protocol. We use the syntax of Maude extended with our constructs for the synchronous composition. Also, we make intensive use of parameterized programming to encapsulate components and specify interfaces. This paper must be considered a companion to some of our previous ones.
## Contents

1. **Introduction**  
   - Page 1

2. **The alternating bit protocol**  
   - Page 1

3. **Common**  
   - 3.1 Stages  
     - Page 2
   - 3.2 Properties  
     - Page 2
   - 3.3 Matchable sorts  
     - Page 3

4. **Interfaces**  
   - 4.1 Producer and consumer  
     - Page 5
   - 4.2 Sender and receiver  
     - Page 6
   - 4.3 Channels  
     - Page 7

5. **Blueprint**  
   - Page 7

6. **Building packets**  
   - Page 9

7. **Implementations**  
   - 7.1 ABP sender  
     - Page 11
   - 7.2 ABP receiver  
     - Page 14
   - 7.3 The channels  
     - Page 15

8. **Final system**  
   - Page 16

9. **Final remarks**  
   - Page 17
1 Introduction

This is a companion paper to our previous [5, 4, 3]. In them, we describe an operation of synchronous composition for rewrite systems and the means to implement and use it in Maude. Some familiarity with those papers is probably needed to understand the present one. This contains an example implementing the alternating bit protocol. Parts of this example are already in [5], but here it is fully detailed and explained.

The example is coded using the syntax of Maude (described, for example, in [1]), extended with the constructs we propose for synchronous composition. Indeed, the code that follows is not executable in the existing implementation of Maude for two reasons. First, the extensions to the language that we propose to support synchronous composition have not been implemented as yet; we intend to do so soon. Second, we make intense use of parameterized programming. Though all of it is theoretically plausible and sticks to the proposals in [2], it is only partially implemented in Maude. (Full Maude, also described in [1], has got parameterized theories and views, but not to the limit that we use them here.)

2 The alternating bit protocol

The alternating bit protocol, or ABP, is used to transmit messages on a lossy channel, that is, a channel that can lose some of the messages it receives before delivering them at the other end. According to ABP, a bit is attached to each packet of information sent through the channel. The sender must keep on sending the same packet with the same bit until it receives an acknowledgement (let’s abbreviate it to ack) from the receiver with that same bit. Then, the sender starts sending the next packet with the bit inverted. As also acks can be lost in the channel, the receiver must keep on acknowledging until it receives a new message, with a different bit, that suffices as proof that its ack was received and processed by the sender.

We consider an ABP system as consisting of six components:

At the ends there are a producer and a consumer, that do not care about communication protocols. The sender and the receiver implement the ABP. There are two channels: one for transmitting messages originating in the producer; the other for transmitting acks. We call message to whatever the producer wants to transmit to the consumer. The pieces of information that are sent through a channel are called packets. It is the sender’s task to transform each message
into one or more packets; the receiver has the inverse task. What precisely
is a packet depends on the protocol used. In our implementation of ABP, we
make the rather unrealistic assumption that we can build channels capable of
transmitting packets of whatever data type. Our implementation of channels is
parametric on that data type.

The six modules in our diagram are not all of the same nature. The producer,
the consumer, and the two channels can be seen as representing physical entities,
while the sender and the receiver may well be pieces of software running on some
of them. This difference, however, has no role in specifications.

3 Common

Some pieces of code are used often in the specification. They are best in a
common module to be imported when and where needed. This section contains
their specifications.

3.1 Stages

Atomic egalitarian modules are the ones that implement basic, non-composed
systems, defining what states and transitions are in a particular component.
The following module is useful to be imported in them:

```
fmod STAGES is
    sorts State Trans Stage .
    subsorts State Trans < Stage .
    op init : -> Stage .
endfm
```

A stage is either a state or a transition. We declare the name `init` for the initial
stage; it can be a transition, as well as a state.

3.2 Properties

Properties are used to formulate syncing criteria. They can be thought of as
ports or handles, different metaphors being appropriate in different cases. They
work like functions that take values at states and at transitions. But we do not
implement them as functions but like this (be aware that we slightly modify
this definition in the next section):

```
fmod PPTY{X :: TRIV} is
    pr MAYBE{X} .
    sort Stage .
    sort Ppty{X} .
    op _@_ : Ppty{X} Stage -> Maybe{X} .
endfm
```

Just as a reminder, the theory `TRIV` is defined like this:

```
fth TRIV is
    sort Elt .
endfth
```
The idea, thus, is that an element of sort \( P\text{pty} \{X\} \) evaluates, through the operator \( \& \), to a value of sort \( X\text{Elt} \) at each stage. Indeed, this is not completely true for two reasons. First, a property may be undefined at some stages (the rationale for this is explained in [4, 3]). Second, the operator \( \& \) is declared as returning a value of sort \( \text{Maybe}\{X\} \), instead of plain \( X\text{Elt} \). The definition of the parametric module \( \text{MAYBE}\{X\} \) is this:

```plaintext
fmod MAYBE\{X :: TRIV\} is
  sort Maybe\{X\} .
  subsort X\text{Elt} < Maybe\{X\} .
  op none : -> Maybe\{X\} .
endfm
```

That is, the sort \( \text{Maybe}\{X\} \) contains all values in the sort of the parameter, \( X\text{Elt} \), plus a dummy value called \texttt{none}. (In the standard implementation of \( \text{MAYBE} \) in Maude’s prelude, this dummy element is called \texttt{maybe}, but the name \texttt{none} fits better for the use we make of it.) For instance, a property called \texttt{messageBeingSent} can be set to \texttt{none} when no value is being sent. Not every property needs to use this extra value, but we include it in the declaration for the cases when it is useful, that in the present paper are many. Finally, being undefined and being defined to \texttt{none} must not be confused: an undefined property does not pose any restriction for syncing. (All properties in our present implementation of ABP are completely defined.)

### 3.3 Matchable sorts

Our methodology mandates that each component system must be thoroughly meaningful by itself. Each has to be specified in such a way that it can be run either isolated or synced. In many cases, this means that a component shows a wildly non-deterministic behavior if run by itself. A receiver, for instance, must be glad to accept any value the sender can send. This could be represented in the receiver’s side by a rewrite rule like this:

\[
s(v, \ldots) \rightarrow s(v', \ldots)
\]

Here, \( s(\ldots) \) is some state term and \( v \) and \( v' \) are values of some sort that are received and stored by the receiver. However, Maude does not accept fresh variables on the right-hand side of a rule, because they are problematic for execution. A solution is to include the new value \( v' \) in a matching condition, where it can be instantiated:

\[
s(v, \ldots) \rightarrow s(v', \ldots) \text{ if } \{v'\} \cup V := \text{set of possible values}
\]

Maude knows how to solve the matching condition, in a non-deterministic way, and assign values to \( v' \) and \( V \).

When the time comes for the composition to be performed, and syncing criteria are specified, the composed, global behavior can become deterministic, if the sender only sends a particular value at any given time, deterministically, as is usual. But if we are interested in executing or analyzing the receiver by
itself, the value of \(v'\) has to be chosen non-deterministically from the “set of possible values”, which, of course, has to be finite for Maude to be able to solve the matching.

For our implementation of ABP here, we have insisted that it be parametric on the sort of messages being interchanged. According to the above, the parameter cannot be just a TRIV, but needs to provide the elements needed for writing and solving matching conditions, among them the set of all possible values for each sort that we want to use for syncing. For those reasons we define this theory:

```plaintext
fth MATCHABLE is
  sorts Elt Elts .
  subsort Elt < Elts .
  op noElts : \to Elts .
  op allElts : \to Elts .
endfth
```

With all the attributes assoc, comm, and id:, the operator & makes Elts equivalent to sets of Elt. This seems appropriate so that we can choose an Elt from an Elts non-deterministically with total freedom. (A technical side note: in Maude, it is not possible to declare a sort Elt and then state that we are going to use SET{Elt} as well. The argument for SET has to be a view, defined outside the current module.)

The constant allElts is interpreted as providing the whole set of possible values of sort Elt. The matching conditions we use will have the form if \(E & EE := \text{allElts}\), for \(E\) and \(EE\) variables of sorts Elt and Elts, respectively.

As a consequence, the sorts returned by a property cannot, in general, be just a TRIV: they need to be a MATCHABLE. So we redefine the module PPTY:

```plaintext
fmod PPTY\{X :: MATCHABLE\} is
  pr MAYBE{Matchable}\{X\} .
  sort Stage .
  sort Ppty\{X\} .
  op _@_ : Ppty\{X\} Stage -> Maybe\{X\} .
endfm
```

To instantiate MAYBE, we are using the fact that a matchable sort can be used whenever a TRIV is expected:

```plaintext
view Matchable from TRIV to MATCHABLE is
  sort Elt to Elt .
endv
```

There are several instances of use of all this below.

The use of MATCHABLE can be seen as a dirty trick, and its need can be seen as an annoying consequence of our choices and Maude’s design. For, suppose the sender and the receiver were specified together. And suppose the sender works in a deterministic way, so that if it has \(v\) stored, then it next sends and stores a new value \(f(v)\), for some function \(f\) implemented as part of the sender’s specification. Then, our rule would look like this:

\[
s(v,\ldots) \rightarrow s(f(v),\ldots) \rightarrow r(f(v),\ldots),
\]
where $s$ represents the sender’s states and $r$ the receiver’s. There is no need for matching and matchable sorts here. But compositionality is lost in this way, both for specification and for verification. Also, there is some conceptually sound truth in the matchable solution. Real-world systems are not able to produce, send or receive data of unbounded size. Any system has its bounds, embedded into their construction. It is appropriate, then, to include such bounds as part of their specifications. The matchable trick provides an explicit solution for that.

4 Interfaces

We use theories (in the parameterized-programming sense) to specify the interfaces for component modules, understanding by this the list of properties they must define and their sorts. All the external world needs to know about an implementation module is that it conforms to the requirements of a given theory.

We are also interested in compositional verification. This is work still to be done, so we do not include here anything related to that. But let us point just that, to that aim, assume and guarantee temporal formulas can be included into a theory (with the needed extensions to Maude’s syntax), with the same aim that nonexec equations are included into theories in standard Maude.

Properties can be used to emulate value passing, as described in [5, 4, 3]. In these cases, they can be understood as directional, that is, as being an out property at the sender side, and an in property at the receiver’s. That is why we have shown directed arrows in the diagram above. However, properties per se are not directional, so that the producer and the consumer conform to the same interface, as do the sender and the receiver. This is exploited below.

One more note, before showing the code for the interfaces. We have decided to make all the ABP system parametric on the sort of the messages interchanged. As shown below, this means that we need to use parameterized theories and views. As noted above, this does not work in the current implementation of Maude (or Full Maude), but we still have preferred to take the chance to show the nice possibilities of the use of parameterized programming in rewriting logic.

4.1 Producer and consumer

Both producers and consumers must conform to an interface showing to the world a port through which messages are synced (sent in one case, received in the other). That is, they must conform to this parameterized theory:

```plaintext
th PROCESS-IF{Msg :: MATCHABLE} is
  pr Ppty{Msg} .
  op msgMoving : -> Ppty{Msg} .
endth
```

We always name interfaces with an ending -IF. We use the name `msgMoving` for the property, to be agnostic about whether it is being sent or received.

Syntactically, this is a functional theory, that is, it does not include any rules. Thus, it could be enclosed between `fth` and `endfth`. But the instantiations of
this theory have to be system modules, that is, modules including rules. So we feel it is more fitting to declare the theory as a system one. The same is true for other theories below.

(As a technical side note, Maude allows a functional theory to be instantiated by a system module. It happens, however, that a functional module parameterized by a functional theory becomes a system module when its parameter is instantiated by a system module.)

The parameter \( \text{Msg} \) represents the sort of the messages the process is able to send/receive. It must be instantiated (when needed) by a module (or, rather, a view) that conforms to the theory \text{MATCHABLE}. Indeed, when the whole system is put in place by syncing all the components, we must ensure that the producer and the consumer conform to \text{PROCESS-IF} instantiated with the same parameter, that is, that the producer produces the same sort of messages that the consumer is able to consume. See below. (To be strict, the sort of the messages produced must be a subsort of the ones that can be consumed.)

### 4.2 Sender and receiver

The sender and the receiver are the only components of the system that are aware of the protocol being used and that are expected to implement the ABP. Because the lack of directionality of properties, they conform to the same interface:

```plaintext
th PROTOCOL-IF{ProcMsg :: MATCHABLE,
Pck2Chnl :: MATCHABLE,
PckFChnl :: MATCHABLE} is
  pr PPTY{ProcMsg} + PPTY{Pck2Chnl} + PPTY{PckFChnl} .
  op procMsgMoving : -> Ppty{ProcMsg} .
  op pckLeaving2Chnl : -> Ppty{Pck2Chnl} .
  op pckComingFChnl : -> Ppty{PckFChnl} .
endth
```

There are three parameters and three properties, that happen to correspond one to one. This correspondence is not a general rule; indeed, interfaces do not even need to be parameterized at all. The first parameter, \( \text{ProcMsg} \), represents the sort of the messages that the sender takes from the producer and that the receiver handles to the consumer. The second parameter, \( \text{Pck2Chnl} \), represents the sort of the packets that the sender puts into the message channel, or the receiver puts into the ack channel, to be delivered at the other side. The third parameter, \( \text{PckFChnl} \), is the sort of packets received from the channel. (We often use 2 as short for to, and F as short for from.) When putting the whole ABP system in place, the sending channel for the sender has to be the same as the receiving channel for the receiver, and vice versa.

It is the sender’s job to split the message from the producer in as many pieces as needed and to transform it into one or more packets. The receiver has the job of decoding one or more packets to recover the message. This interface, as it stands, is valid for modules implementing any ack-based protocol, not just ABP. When putting the whole ABP system together, instantiations of some of
these parameters must coincide, to be coherent with the lines in the diagram above.

4.3 Channels

The complete ABP system uses two channels, one for message packets, the other for ack ones. Except for the sort of the packets sent through them, the interface for both channels is the same. Even the implementation of the inner workings of the channels can be the same, provided it is parametric on the sort of packets. This is the interface:

```plaintext
th CHANNEL-IF{Pck :: MATCHABLE} is
   pr PPTY{Pck}.
   ops pckComing pckLeaving : -> Ppty{Pck}.
endth
```

A channel can lose some of the packets that arrive to it, but it must be granted that a packet repeatedly put into the channel eventually reaches the other end. This kind of temporal properties can be added to a theory as semantic requirements, and would also help in compositional verification, assume-guarantee style. As already mentioned, this is pending work.

5 Blueprint

We call blueprints to the recipes that specify how to assemble component systems to build a composed one. Blueprints are coded as parameterized modules that receive as parameters the components, conforming to appropriate theories. The blueprint’s job is to specify syncing criteria.

We have preferred not to assemble all the six components of the system in one whole unit. Instead, we assemble only the sender, the receiver, and the two channels, to produce a communication system to which a producer and consumer can be attached later. This is the blueprint for such a composition:

```plaintext
emod COMM-SYSTEM-BP
   { Sndr :: PROTOCOL-IF{Msg :: MATCHABLE,
                           MsgPck :: MATCHABLE,
                           AckPck :: MATCHABLE},
     MsgChnl :: CHANNEL-IF{MsgPck :: MATCHABLE},
     AckChnl :: CHANNEL-IF{AckPck :: MATCHABLE},
     Rcvr :: PROTOCOL-IF{Msg :: MATCHABLE,
                           AckPck :: MATCHABLE,
                           MsgPck :: MATCHABLE}
   } is
   sync Sndr || MsgChnl || AckChnl || Rcvr
      on Sndr$pckLeaving2Chnl = MsgChnl$pckComing
      /\ MsgChnl$pckLeaving = Rcvr$pckComingFChnl
      /\ Rcvr$pckLeaving2Chnl = AckChnl$pckComing
      /\ AckChnl$pckLeaving = Sndr$pckComingFChnl.
endem
```

There is much to be explained here. First, the module is enclosed between the keywords `emod` and `endem`. The `e` is for egalitarian. This is a new kind of
module that we propose, not present in standard Maude. Egalitarian modules are expected to include a \textit{sync on} instruction. They must \textit{protect} the result of the syncing, that is, they must not contain new rules, nor add new states or transition terms, nor make existing ones become equal. The only extra code that an \textit{emod} is allowed to contain is whatever may be required for the definition of properties of the composed system, in case it is going to be used as a component in turn. We illustrate such an extreme in Section 8.

Each of the four parameters in \texttt{COMM-SYSTEM-BP} implements a theory that, in turn, is parameterized. All nested parameters have to be made explicit. Coincidence of names for parameters represents shared parameters. It is the case, for example, with \texttt{Msg}, that is a parameter of \texttt{Sndr} and \texttt{Rcvr}, and needs to be shared, as it represents the sort of the messages they interchange.

The \textit{sync on} instruction shows which systems must be synced and with which criteria. We stick to a methodology according to which all modules in a \textit{sync on} instruction must be among the parameters of the \textit{emod} (this is not mandatory, however: they can be any modules already defined). Each criterion is a condition that must be satisfied when the properties are evaluated at the component stages visited at each given time.

The four criteria in our \textit{on} clause nicely correspond to the four arrows in the diagram above. Each criterion tells that the value that is leaving a component is arriving to another.

For our implementation to work, each of these properties is expected to be defined at all states and transitions, with value \texttt{none} when no data interchange is taking place. In this way, each criterion ensures value-passing and simultaneity at the same time.

We use the syntax with the $\$-$ symbol to make it clear to which component system each property belongs. This syntax is already used in Maude to qualify sort names from parameters. In standard Maude, operators need not be qualified, because the sorts of the operands are enough to disambiguate them. However, in our setting, we prefer to avoid mentioning stages explicitly in our criteria. So, we need a means to disambiguate.

I want to spend a few more lines on this. We have tried some other possible syntaxes for writing criteria. For instance:

```maude
emod M{M1 :: T1, M2 :: T2} is
  sort Stage .
  var G : Stage .
  sync M1 || M2
  on P1 @ M1(G) = P2 @ M2(G) .
endem
```

We use the name of the parameter modules as projection operators. The variable $G$ is for the global, composed stage. Thus, it is explicit where each property is to be evaluated, and the $\$-$syntax is not needed. This possibility, however, has a formal and a conceptual problem. The formal one is that the sort \texttt{Stage}, referred to the global, composed stages, can only be in existence after the \textit{sync on} instruction, probably produced by it as a side effect. It is odd to declare \texttt{Stage} and $G$ before that. The conceptual problem is that we prefer not to mention
the global stage, or even think about it. Because, what is the global stage of the system composed by Google’s server and my browser? It does not matter, and it does not help to think of a global stage. We only need to make sure that the different components sync as appropriate. We concede that not all examples are as distributed as web browsing, but we still think it is better to avoid mentioning explicitly global stages.

6 Building packets

The setting above does not require any relation between messages (what the producer and the consumer need to interchange) and packets (what the channels are able to transmit). But they are certainly not independent. As a first approximation, we define next a module for building packets from some contents (a part of a message, for example) and a wrapper (a Boolean representing an alternating bit, in our case). It is important that both arguments are MATCHABLE, and that the result is as well.

```plaintext
fmod PACKET-BUILDER{Cnt :: MATCHABLE, Wrp :: MATCHABLE} is
  sorts Packet{Cnt, Wrp} Packets{Cnt, Wrp} .
  subsort Packet{Cnt, Wrp} < Packets{Cnt, Wrp} .
  op packet : Cnt$Elt Wrp$Elt -> Packet{Cnt, Wrp} .
  op noPackets : -> Packets{Cnt, Wrp} .
  op _&_ : Packets{Cnt, Wrp} Packets{Cnt, Wrp}
        -> Packets{Cnt, Wrp} [comm assoc id: noPackets] .
  op allPackets : -> Packets{Cnt, Wrp} .

  var C : Cnt$Elt .
  var W : Wrp$Elt .
  var CC : Cnt$Elts .
  var WW : Wrp$Elts .
  eq allPackets = cartesianProd(allElts.Cnt$Elts, allElts.Wrp$Elts) .

  op cartesianProd : Cnt$Elts Wrp$Elts -> Packets(Cnt, Wrp) .
  eq cartesianProd(noElts, WW) = noPackets .
  eq cartesianProd(C & CC, WW) = cartesianProdAux(C, WW)
    & cartesianProd(CC, WW) .
  eq cartesianProdAux(C, noElts) = noPackets .
  eq cartesianProdAux(C, W & WW) = packet(C, W)
    & cartesianProdAux(C, WW) .
endfm
```

view Packet{Cnt :: MATCHABLE, Wrp :: MATCHABLE}
  from MATCHABLE
  to PACKET-BUILDER{Cnt, Wrp} is
  sort Elt to Packet{Cnt, Wrp} .
  sort Elts to Packets{Cnt, Wrp}.
  op noElts to noPackets .
  op _&_ to _&_ .
  op allElts to allPackets .
endv

In our case, wrappers are always Booleans, so we need to proof they are matchable:
view MatchableBool from MATCHABLE to BOOL + SET{Bool} is
  sort Elt to Bool.
sort Elts to Set{Bool}.
op noElts to empty.
op _ & _ to _,_.
op allElts to term (false, true).
endv

The view Bool is the standard one from TRIV to BOOL, and is defined in Maude’s prelude. We need to use Booleans and sets of Booleans together; that’s why the destination of our view is BOOL + SET{Bool}. (A quite technical side note: the MATCHABLE theory requires subsort Elt < Elts. The standard implementation of SET, in Maude’s prelude, includes subsort X$Elt < NeSet{X} < Set{X}, so everything works. But it is implementation dependent. That is, if SET were implemented using a constructor to transform an element into a singleton set, the view MatchableBool, as coded above, would not work.)

Packets sent from the receiver to the sender (that is, acks) consist, in our implementation, of the alternating bit plus a contents consisting on an ack mark. We need this:

fmod ACK is
  sort Ack.
op ack : -> Ack.
endfm

view Ack from TRIV to ACK is
  sort Elt to Ack.
endv

view MatchableAck from MATCHABLE to ACK + SET{Ack} is
  sort Elt to Ack.
sort Elts to Set{Ack}.
op noElts to empty.
op _ & _ to _,_.
op allElts to ack.
endv

A concrete implementation of messages is missing, because we still insist that all our implementation is parametric on the sort of messages. So we assume that a module MSG and a view MatchableMsg have been defined, or are going to be defined when needed. With that, the two views that we are going to use are

Packet{Msg, MatchableBool}
Packet{MatchableAck, MatchableBool}

That means, in particular, that a message packet contains a whole message in one piece (in addition to the alternating bit).

7 Implementations

We show next possible implementations of the ABP sender, the ABP receiver, and the channels. As announced above, the two channels work the same, just with different arguments for the contents of packets, so only one implementation is needed for them.
7.1 ABP sender

It is often useful to identify the internal modes in which a system can be, and use them as part of the state terms and transition terms of the system. For the ABP sender, we use three state modes:

- **ready2Send**: ready to send a packet through the message channel,
- **waiting4Ack**: waiting for an ack to arrive through the ack channel, and
- **ready4Msg**: ready to receive a new message from the producer.

For transitions, we use three modes as well:

- **takingMsg**: taking a new message from the producer,
- **sending**: sending a packet through the message channel, and
- **receivingAck**: receiving a packet from the ack channel.

We tend to give transition modes names ending in, or containing, *ing* (in this particular case, also a state bears such a name).

With these modes, the workings of the sender can be pictured like this:

The flow from mode to mode is rather deterministic, except in two cases: after having received an ack, depending on the value of the alternating bit, we may need to keep waiting for an appropriate bit or to go on for the next message from the producer; and we can exit a **waiting4Ack** state either because we have indeed received an ack (be it valid or not) or because, tired of waiting, we decide to send our message one more time.

Apart from the mode, a state or transition term needs to include information about the internal configuration, or internal memory, of the system. In our case, the internal configuration of the sender may include the last message taken from the producer, the value of the alternating bit currently in use, the messages or packets ready to be interchanged, and so on. Many of these values would be **none** some time, even most of the time, but this is no problem. However, it seems appropriate to keep the number of variables to a small decent amount.
Our packets are built from messages and bits, and these two pieces of data are almost enough. These two values alone are enough to deduce the whole internal configuration of the sender, except in one case: when the sender is in the course of receiving an ack, the bit coming and the bit last sent have to be kept both, for comparison purposes. We decide, in this example, to use the same three data all the time, even though some of them are going to be none most of the time. See below how all this translates into code.

In the implementation of the sender that follows, each transition involves some interchange of data with other components. This seems to be a useful pattern, but it is not necessary: there may be internal transitions and also interchanging states. This is our implementation:

```
=aemod ABP-SENDER{Msg :: MATCHABLE} is
pr STAGES .
pr MAYBE{Matchable}{Msg} .
pr MAYBE{Bool} .
sorts StateMode TransMode Config .
ops ready2Send waiting4Ack ready4Msg : -> StateMode .
ops takingMsg sending receivingAck : -> TransMode .
op (_,-,-) : MAYBE{Matchable}{Msg} Bool MAYBE{Bool} -> Config .
op (_,-) : StateMode Config -> State .
op (_,-) : TransMode Config -> Trans .
var M : Msg$Elts .
var MM : Msg$Elts .
var B : Bool .
var C : Config .
crl [(takingMsg, (M, B, none))] :
  (ready4Msg, (none, B, none)) => (ready2Send, (M, B, none))
  if M & MM := allElts .
rl [(sending, C)] :
  (ready2Send, C) => (waiting4Ack, C) .
rl [(sending, C)] :
  (waiting4Ack, C) => (waiting4Ack, C) .
rl [(receivingAck, (M, B, not B))] :
  (waiting4Ack, (M, B, none)) => (waiting4Ack, (M, B, none)) .
rl [(receivingAck, (M, B, B))] :
  (waiting4Ack, (M, B, none)) => (ready4Msg, (none, not B, none)) .
eq init = (ready4Msg, (none, true, none)) .
=endaem
```

This module is enclosed between `aemod` and `endaem`. The `e` in those keywords is for egalitarian, as above; the `a` is for atomic, that is, not composed. Atomic egalitarian modules are the ones used to implement basis systems. These are the only ones allowed to contain rules, and the definition of states and transitions. Syntactically, `aemod` is very similar to a standard system module, with the main difference that rule labels can be terms of any complexity.

The internal configuration of the sender is given by a triplet:

```
| MAYBE{Matchable}{Msg} Bool MAYBE{Bool}
```

The only piece of data that is always available is the value of the alternating bit currently in use. It is set to true in the initial state init, and after that, it always has an actual Boolean value. The other two pieces of data, the message being processed and the bit being received, are sometimes none, and we need to
declare the parameters as `Maybe{...}`. The module `MAYBE` expects a `TRIV`, but
we need to feed it a `MATCHABLE`; that is why we need the partially instantiating
view `Matchable).

The rules represent the mode transitions shown in the diagram, together
with their associated changes in the internal configuration.

In the definition of the `init` stage, we set the initial message to `none`. It
relies on this data being of a `maybe` sort, guaranteed to include the special value.
Otherwise, the initial stage could not have being defined so easily, because it
would depend on the particular instantiation of the parameter `Msg`.

This implementation needs some fairness conditions to work properly. Other-
wise, the sender may have an ack available but ignore it and keep on sending
the same message with the same bit, preventing the whole system from evolv-
ing. The same happens to the implementation of the receiver, and to that of
the channels, both below. Fairness conditions like these ones are not possible to
add as code within the implementation. Again, temporal properties like these
ones, representing semantic requirements, can be added to our theories.

For this implementation to actually conform to `PROTOCOL-IF`, we need to
define properties. In this case, it even seems the implementation is not complete
without the properties, as their values tell us what they are ready to receive or
send through their `ports`. Properties for syncing are usually defined in a module
extending the one that specifies the inner workings of the system.

```plaintext
aemod ABP-SENDER-PPT{Msg :: MATCHABLE} is
    pr ABP-SENDER{Msg}.
    pr PACKET-BUILDER{Msg, MatchableBool}.
    pr PACKET-BUILDER{MatchableAck, MatchableBool}.
    pr PPTY{Msg}.
    pr PPTY{Packet{Msg, MatchableBool}}.
    pr PPTY{Packet{MatchableAck, MatchableBool}}.
    op procMsgMoving : -> Ppty{Msg}.
    op pckLeaving2Chnl : -> Ppty{Packet{Msg, MatchableBool}}.
    op pckComingFChnl : -> Ppty{Packet{MatchableAck, MatchableBool}}.
    var G : Stage.
    var M : Msg$Elt.
    vars B B' : Bool.
    eq procMsgMoving @ (takingMsg, (M, B, none)) = M.
    eq procMsgMoving @ G = none.
    eq pckLeaving2Chnl @ (sending, (M, B, none)) = packet(M, B).
    eq pckLeaving2Chnl @ G = none.
    eq pckComingFChnl @ (receivingAck, (M, B, B')) = packet(ack, B').
    eq pckComingFChnl @ G = none.
denaem
```

The view is now easy:

```plaintext
view AbpSender{Msg :: MATCHABLE}
    from PROTOCOL-IF{Msg,
        Packet{Msg, MatchableBool},
        Packet{MatchableAck, MatchableBool}}
    to ABP-SENDER-PPT{Msg} is
    op procMsgMoving to procMsgMoving.
    op pckLeaving2Chnl to pckLeaving2Chnl.
    op pckComingFChnl to pckComingFChnl.
```
This view only needs a parameter to implement a theory with three, because the other two parameters are built from the first using the PACKET-BUILDER tool define earlier.

7.2 ABP receiver

The workings of the receiver are similar to the ones for the sender. We hope the diagram of modes and the code are now easy to understand with no further remarks.

```
aemod ABP-RECEIVER{Msg :: MATCHABLE} is
  pr STAGES .
  pr MAYBE{Matchable}{Msg} .
  pr MAYBE{Bool} .
  sorts StateMode TransMode Config .
  ops ready2SendAck waiting4Pck pckReceived : -> StateMode .
  ops givingMsg sendingAck receivingPck : -> TransMode .
  op (_,_,_) : MAYBE{Matchable}{Msg} MAYBE{Matchable}{Msg} MAYBE{Bool} MAYBE{Bool} -> Config .
  op (_,_) : StateMode Config -> State .
  op (_,_) : TransMode Config -> Trans .
  var M : Msg$Elt .
  var MM : Msg$Elt .
  var B : Bool .
  var C : Config .
  rl [(givingMsg, (M, B, none))] :
    (pckReceived, (M, B, none)) => (ready2SendAck, (none, B, none)) .
  rl [(sendingAck, C)] :
    (ready2SendAck, C) => (waiting4Pck, C) .
  rl [(sendingAck, C)] :
    (waiting4Pck, C) => (waiting4Pck, C) .
  crl [(receivingPck, (M, B, B))] :
    (waiting4Pck, (none, B, none)) => (waiting4Pck, (none, B, none))
    if M & MM := allElts .
  crl [(receivingPck, M, B, not B)] :
    (waiting4Pck, (none, B, none)) => (pckReceived, (M, not B, none))
    if M & MM := allElts .
  eq init = (waiting4Pck, (none, false, none)) .
endaem
```
7.3 The channels

The implementation of a channel is quite straightforward. It can just accept a packet, deliver it, or lose it. We could use here the same modes thing as above, but it doesn’t pay off for this simple system.

```plaintext
aemod CHANNEL{Pck :: MATCHABLE} is
  pr STAGES .
  pr MAYBE{Matchable}{Pck} .
  subsort May{Matchable}{Pck} < State .
  ops acceptingPck deliveringPck : Pck$Elt -> Trans .
  op losingPck : -> Trans .
  var P : Pck$Elt .
  var PP : Pck$Elts .
  crl [acceptingPck(P)] : none => P if P & PP := allElts .
  rl [deliveringPck(P)] : P => none .
  eq init = none .
endaem
```

```plaintext
aemod CHANNEL-PPT{Pck :: MATCHABLE} is
  pr CHANNEL{Pck} .
  pr STAGES .
  pr FPTY{Pck} .
```
op pckComing pckLeaving : -> Ppty{Pck}.
var P : Pck$Elt.
var G : Stage.
eq pckComing @ acceptingPck(P) = P.
eq pckComing @ G = none.
eq pckLeaving @ deliveringPck(P) = P.
eq pckLeaving @ G = none.
endaem

view Channel{Pck :: MATCHABLE}
  from CHANNEL-IF{Pck}
to CHANNEL-PPT{Pck} is
  op pckComing to pckComing .
  op pckLeaving to pckLeaving .
endv

8 Final system

With all the components implemented, it only remains to feed them to the blueprint to obtain the ABP system. We prefer to let the sort of messages as a parameter until the end:

emod ABP-SYSTEM{Msg :: MATCHABLE} is
  pr COMM-SYSTEM-BP{AbpSender{Msg},
    Channel{Packet{Msg, MatchableBool}},
    Channel{Packet{MatchableAck, MatchableBool}},
    AbpReceiver{Msg}} .
endaem

Before finishing, it is interesting to note that any implementation of COMM-SYSTEM-BP can be viewed as a channel—a kind of trustworthy channel. We only need to make explicit the properties:

emod COMM-SYSTEM-BP-PPT
  { Sndr :: PROTOCOL-IF{Msg :: MATCHABLE,
    MsgPacket :: MATCHABLE,
    AckPacket :: MATCHABLE},
    MsgChnl :: CHANNEL-IF{MsgPacket :: MATCHABLE},
    AckChnl :: CHANNEL-IF{AckPacket :: MATCHABLE},
    Rcvr :: PROTOCOL-IF{Msg :: MATCHABLE,
    AckPacket :: MATCHABLE,
    MsgPacket :: MATCHABLE} }
  is
  pr COMM-SYSTEM-BP{Sndr, MsgChnl, AckChnl, Rcvr} .
  pr PPTY{Msg} .
  ops msgComing msgLeaving : -> Ppty{Msg} .
  var G : Stage .
  eq msgComing @ G = procMsgMoving @ Sndr(G) .
  eq msgLeaving @ G = procMsgMoving @ Rcvr(G) .
endaem

This also allows us to show how properties of a composed system are defined in terms of the properties of the components. The sort Stage for the composed system is assumed to be declared and defined (in a tuple-like way) as a side
effect of the sync on instruction. The same for the projection operators with
the names of the component systems.

We are using here the global stage of the composed system, against which
we argued in Section 5. It is difficult to avoid it, and it can be justified now:
if the composed system is going to be used as a component in turn, it is an
indication that it has a kind of unity, a complete existence as a system.

With these properties, the following shows how the system can be viewed as
a channel:

```plaintext
view CommSystemAsChannel{Msg :: MATCHABLE}
  from CHANNEL-IF{Msg}
  pr COMM-SYSTEM-BP{AbpSender{Msg},
    Channel{Packet{Msg, MatchableBool}},
    Channel{Packet{MatchableAck, MatchableBool}},
    AbpReceiver{Msg}
  } is
    op pckComing to msgComing .
    op pckLeaving to msgLeaving .
endv
```

9 Final remarks

Unfortunately, as already mentioned, most of the code in this paper does not
run on the current implementations of Maude and Full Maude. We can make
do without nested parameters by using the poor man’s parameterization trick:
importing the implementation module msg.maude into any module that would
instead take Msg as parameter. This is certainly not perfect. A complete im-
plementation of parameterized programming in Maude would be great, but it
remains to be seen if it will be available at some future time.

We take as our job to implement in the near future the other missing ingredi-
ent, that is, everything that is needed to perform the synchronous composition
in Maude. Our aim is to make executable a specification equivalent to the one
given up here.
References


