Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008
© 2008 The authors and IOS Press. All rights reserved.

RRABP: Point-to-Point Communication over Unreliable Components

Bernhard H.C. SPUTH, Oliver FAUST and Alastair R. ALLEN
School of Engineering, University of Aberdeen, Aberdeen, AB24 3UE, UK
bernhard@erg.abdn.ac.uk, {o.faust, a.allen}@abdn.ac.uk

Abstract. This paper establishes the security, stability and functionality of the resettable receiver alternating bit protocol. This protocol creates a reliable and blocking channel between sender and receiver over unreliable non-blocking communication channels. Furthermore, this protocol permits the sender to be replaced at any time, but not under all conditions without losing a message. The protocol is an extension to the alternating bit protocol with the ability for the sender to synchronise the receiver and restart the transmission. The resulting protocol uses as few messages as possible to fulfil its duty, which makes its implementation lightweight and suitable for embedded systems. An unexpected outcome of this work is the large number of different messages needed to reset the receiver reliably.

Keywords. point-to-point communication, reliability, unstable sender.

Introduction

Over the past years CSP process networks have become more and more dynamic. Such dynamic process networks adapt to a changing environment during runtime. The start of this dynamicalisation of process networks was the introduction of mobile channels and mobile processes to occam-π [1,2]. Mobile channels reconfigure the network connections. Similarly, mobile processes reconfigure the functionality of a process network. Both techniques allow the designer to influence the network functionality during runtime. These mobility techniques have since been added to the Communicating Sequential Processes for Java (JCSP) library, by the jcsp.mobile package [3]. This package takes mobility a step further by making processes and channels mobile over the nodes in a TCP/IP network.

Another technique to adjust process networks to changes in the environment is process subnetwork termination by localised graceful-termination [4], also known as localised poisoning. It is no problem to come up with scenarios where a change in the environment renders a complete subnetwork obsolete. One example where subnetwork termination is helpful is a network server. Such as server assigns each connecting client a dedicated process subnetwork. If a client disconnects from the server, its dedicated process subnetwork becomes obsolete. It is highly desirable to terminate these subnetworks, because this releases resources and prevents errors. Localised poisoning means that the termination message (poison) does not leave the subnetwork to be terminated. This containment of poison within a subnetwork makes it possible to terminate clearly defined parts of a process network. The localised poisoning technique was first presented for JCSP in the form of JCSP-Poison [5]. Since then it has been refined and integrated into the JCSP library [6]. The ability to perform a localised graceful-termination is also available in C++CSP [7]. Another approach to terminating subnetworks and even replacing them is the exception handling mechanism in Communicating Threads for Java (CTJ) [8].

Both techniques, mobility and poisoning rely on an entity controlling or initiating the process network change. This controlling entity ensures that the environment outside the terminating process subnetwork knows about this termination and can prepare for it. However, there are situations in which no entity controls the change of the process network. This is, for instance, the case when a process network is spread over multiple unreliable nodes, which communicate over unreliable communication channels. In such systems, nodes may suddenly decide to terminate and then restart later. Furthermore, unreliable communication channels lose and replicate messages in an unpredictable way. An example for such a system is the water quality monitoring system of the WARMER (**WA**ter **R**isk **M**anagement in **E**u**R**ope) consortium [9,10]. This system consists of multiple in-situ monitoring stations (IMS) and one data centre. The IMSs periodically transmit their measurement data to the data centre, where the data gets accumulated and stored. There are multiple unreliable components involved in this system. The first unreliable component is the communication channel between the IMSs and the data centre. Apart from dropping completely, this unreliable communication channel has other undesirable properties such as losing or replicating data-messages which an IMS entrusts to it. Furthermore, the individual IMS might be unstable, for instance it runs out of energy, or is severely damaged or sunk due to harsh weather conditions. The data centre has to be classified as unstable as well, because its hardware may fail.

The unreliable channel does not pose a very big problem, because there are protocols which can deal with this, such as Bartlett *et. al.* Alternating Bit Protocol (ABP) [11]. However, the ABP does not handle unstable senders and receivers. An unstable receiver may cause a message duplication, while an unstable sender can cause a message loss. A message duplication is unpleasant, but can be lived with and on the receiver side even dealt with, because the receiver is aware that it just started. However, a potential message lost is clearly unacceptable.

The Resettable Receiver Alternating Bit Protocol (RRABP) solves the problems which unstable senders introduce. This is achieved by extending the ABP with a mechanism with which the sender can reset the receiver and thus avoid the potential message loss. This was achieved by introducing three new messages, instead of only a single reset message as we anticipated. However, model checking revealed undesired resets of the receiver and message losses when only a single reset message was in use.

The next section gives an introduction to the Alternating Bit Protocol, which is the foundation to the Resettable Receiver Alternating Bit Protocol. Section 2 develops the formal specification of the protocol, followed by the formal model of the RRABP in Section 3. Section 4 establishes security, stability, and functionality of the protocol. The paper closes with conclusions and further work.

1. Materials and Methods

The RRABP is designed as an extension of the alternating bit protocol. This section explains how the ABP overcomes unreliable communication channels. After detailing the ABP this section gives a model for an unreliable communication channel. These details are based upon Roscoe's description and model of the alternating bit protocol in [12, page 130ff].

1.1. Alternating Bit Protocol

The alternating bit protocol is designed for point-to-point communication systems like the one illustrated in Figure 1. This system consists of the sender *S*, the unreliable bidirectional communication channel *CHAN* and the receiver *R*. To overcome the unreliable communication channel, *S* appends a tag-bit to each data-message, input from the channel *in*. This tag-bit alternates between 0 and 1 for each data-message. This allows *R* to identify replications

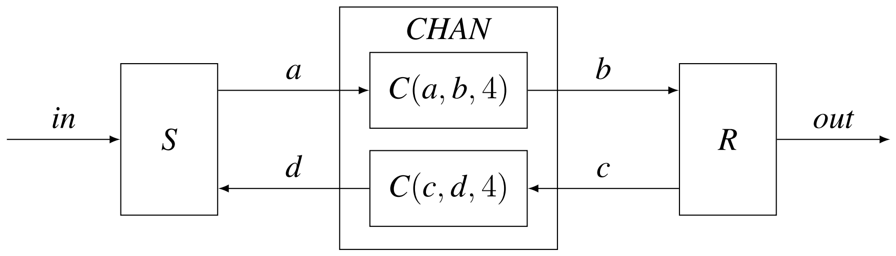

Figure 1. Process network structure used to model a point-to-point communication system.

of the data-message caused by the channel. To counter the loss of data-messages introduced by the unreliable communication channel, the sender *S* sends multiple copies of each data-message until it receives an acknowledgement from *R*, with the correct tag-bit. *R* has to cope with *CHAN* losing acknowledgments and therefore sends acknowledgements for the last received data-message until it receives a new data-message. The resulting protocol uses as few different messages as possible to fulfil its duty, which makes its implementation lightweight and suitable for embedded systems. An unexpected outcome of this work is the large number of messages necessary to reliably reset the receiver. This has far reaching consequences, because we can show that a certain complexity is necessary to meet the specification. All models or implementations which are less complex do not meet the specification.

1.2. Modelling an Unreliable Channel

The unreliable bidirectional communication channel *CHAN* (Equation 1) consists of two unreliable communication channels $C(i,o,n)$ (Equation 2). These processes input messages from the channel i and may output them on the channel o. The parameter n defines the maximum length of a burst error, *i.e.* how many consecutive messages get dropped (lost) at most and the maximum number of additional copies of the original message. In this paper the maximum burst error length is set to four. This means that the protocol must be able to deal with at most four lost messages in a row, or four duplications of a message.

$$CHAN = C(a,b,4) \; ||| \; C(c,d,4) \tag{1}$$

with

$$\begin{aligned}
C(i,o,n) &= C'(i,o,n,n)\\
C'(i,o,n,r) &= i?x \rightarrow C''(i,o,x,n,r)\\
C''(i,o,x,n,r) &= \left(\begin{array}{l}\textbf{if } r = 0 \textbf{ then}\\ \quad o!x \rightarrow C'(i,o,n,n)\\ \textbf{else}\\ \quad C'''(i,o,x,n,r)\end{array}\right)\\
C'''(i,o,x,n,r) &= o!x \rightarrow C'(i,o,n,n)\\
&\quad \sqcap\, o!x \rightarrow C''(i,o,x,n,r-1)\\
&\quad \sqcap\, C'(i,o,n,r-1)
\end{aligned} \tag{2}$$

1.2.1. Properties of CHAN

We expect the unreliable communication channel *CHAN* (Equation 1) to be non-deterministic, but deadlock and livelock free. The FDR [13] output shown in Figure 2 shows that our expectations are correct. The non-determinism of the process is caused by the fact that the process chooses internally whether it drops or replicates a message, unless it has reached the maximum number of message drops or replications, upon which it must send the current message and then reset its drop / replication counter.

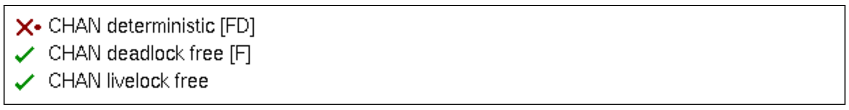
```
✗ CHAN deterministic [FD]
✓ CHAN deadlock free [F]
✓ CHAN livelock free
```

Figure 2. FDR output after checking determinism, deadlock and livelock properties of *CHAN*.

2. The Resettable Receiver Alternating Bit Protocol Specification

The Resettable Receiver Alternating Bit Protocol (RRABP) uses the idea of a tag-bit, introduced by the alternating bit protocol, to overcome the undesirable properties of the unreliable communication channel. Furthermore, it is a sender driven protocol which means that the receiver generates acknowledgements only as a response to a message received from the sender. This prevents surplus messages in case this acknowledgement has been received already, but the sender has no new data-messages to send.

The RRABP ensures that the first message sent by an unreliable sender, *i.e.* a sender which suddenly fails and is subsequently replaced, is not lost. In a system which uses the ABP protocol the first sent data-message gets lost when the receiver classifies it as a duplication of a previous data-message. This happens when the replaced sender uses the same value for the tag-bit which was used by the last transmitted data-message. Clearly, this is undesirable, therefore a protocol extension is necessary which allows the sender to inform the receiver about its replacement. The RRABP achieves this functionality by resetting the receiver. The sender performs this receiver reset during its initialisation. Resetting the receiver forces it into a predefined state, *i.e.* the tag-bit is set to its default value.

The protocol must ensure that no messages are duplicated or lost, when the unreliable sender fails and subsequently is replaced. However, there are two conditions in which it is out of the hands of the protocol and data-messages may be lost nevertheless:

1. The sender dies before it could send the data-message to the receiver. Hence, the message is definitely lost.
2. The sender dies after sending the data-message but before receiving an acknowledgment for the message. This does not necessarily mean that the message was not received by the receiver, but the message may have been lost by the communication channel! It is hard if not impossible to handle this, because the new instance of the sender does not know about the last message which its predecessor tried to send.

Both instances could be handled by introducing a reliable process in front of the sender which waits for an acknowledgment by the sender before supplying a new message to the sender process. However, in the scenario where a complete IMS loses power, this process would terminate as well, making this process unreliable as well.

This section concentrates on showing that the RRABP works in two scenarios:

1. Normal operation: sender and receiver are stable. The corresponding specification is the process *COPY*.
2. Unstable sender together with a stable receiver, with the corresponding specification *SD_SPEC*, where *SD* stands for 'Sender Dies'.

2.1. Normal Operation

The specification model for normal operation behaves like the so called *COPY* process (Equation 3). This means every message on the channel *in* gets output on channel *out*.

$$COPY = in?x \rightarrow out!x \rightarrow COPY \qquad (3)$$

✓ COPY deterministic [FD]
✓ COPY deadlock free [F]
✓ COPY livelock free

Figure 3. FDR output after checking determinism, deadlock and livelock properties of *COPY*.

2.1.1. Properties of COPY

We expect the specification model for normal operation (*COPY*) to be deterministic, deadlock and livelock free. The FDR output of Figure 3 shows that our expectations are correct.

2.2. Unreliable Sender

The RRABP ensures that the receiver keeps the first data-message sent by a replaced sender. Furthermore, once the restart has been completed the system should not lose or duplicate any data-messages.

SD_SPEC (*SD* stands for Sender Dies), Equation 4, specifies how the communication system should behave when a sender suddenly dies. The process is a parallel composition of the sender specification (process *SD_S_SPEC*, Equation 6) and the receiver specification (process *SD_R_SPEC*, Equation 9). The *sender_dies* event represents a failure of the sender, which results in a replacement of the sender. The specification exposes this event to the environment for refinement checks with the implementation model presented in Section 3.3. Figure 4 illustrates the relationship between *SD_S_SPEC* and *SD_R_SPEC* processes. The two processes exchange messages over the channel *tc*. They synchronise on the *ready* event: this event represents a successful synchronisation between sender and receiver processes. Furthermore, they synchronise on the event *sender_dies*, for reasons explained in the discussion of the receiver side specification in Section 2.2.2.

$$SD\_SPEC = SD\_S\_SPEC \underset{\alpha SD\_SYNC}{\|} SD\_R\_SPEC \setminus \{\mid ready, tc \mid\} \tag{4}$$

with

$$\alpha SD\_SYNC = \{\mid ready, tc, sender\_dies \mid\} \tag{5}$$

2.2.1. Sender Side Specification

Process *SD_S_SPEC* (Equation 6) represents, together with *SD_S_SPEC′* (Equation 7) and $SD\_S\_SPEC''(m)$ (Equation 8), the replaceable sender. Initially, *SD_S_SPEC* waits for the occurrence of a *ready* event, or a *sender_dies* event. The *ready* event signals that the receiver accepts data-messages now. After synchronising on the *ready* event, the process transits to the state *SD_S_SPEC′* (Equation 7).

$$\begin{aligned} SD\_S\_SPEC = {} & ready \rightarrow SD\_S\_SPEC' \\ & \quad \Box\ sender\_dies \rightarrow SD\_S\_SPEC \end{aligned} \tag{6}$$

SD_S_SPEC′ accepts the events: *sender_dies* and *in.m*. When it encounters *sender_dies* it transits to state *SD_S_SPEC*, to model the restart of the sender. In case of $in?m$ the process reads in the message from the channel and then transits to the state $SD\_S\_SPEC''(m)$.

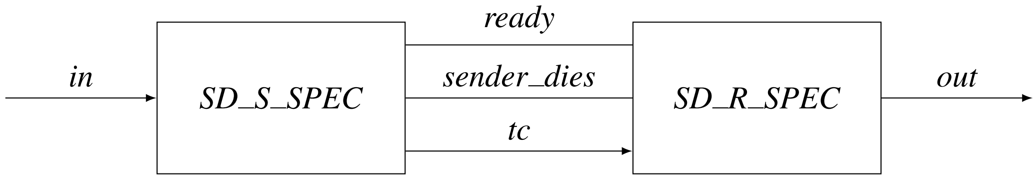

Figure 4. The *SD_SPEC* process detailled

$$\begin{aligned} SD\_S\_SPEC' = {} & in?m \rightarrow SD\_S\_SPEC''(m) \\ & \square\ sender\_dies \rightarrow SD\_S\_SPEC \end{aligned} \tag{7}$$

$SD\_S\_SPEC''(m)$ offers to output the message m on channel tc, after which it transits to SD_S_SPEC. When it encounters *sender_dies* it transits to SD_S_SPEC, losing its message. This models the replacement of the sender.

$$\begin{aligned} SD\_S\_SPEC''(m) = {} & tc!m \rightarrow SD\_S\_SPEC \\ & \square\ sender\_dies \rightarrow SD\_S\_SPEC \end{aligned} \tag{8}$$

The sender side specification is now complete.

2.2.2. Receiver side specification

SD_R_SPEC (Equation 9) is the entry point of the receiver specification. Unless the event *sender_dies* occurs, this process signals that it is *ready* and then transits to state SD_R_SPEC' (Equation 9).

$$\begin{aligned} SD\_R\_SPEC = {} & ready \rightarrow SD\_R\_SPEC' \\ & \square\ sender\_dies \rightarrow SD\_R\_SPEC \end{aligned} \tag{9}$$

SD_R_SPEC' offers to read a data-message from channel tc. After this it transits to $SD\_R\_SPEC''(m)$ (Equation 11). If instead *sender_dies* occurs the process transits to SD_R_SPEC. This models the case that the receiver expects to be reset by the sender.

$$\begin{aligned} SD\_R\_SPEC' = {} & tc?m \rightarrow SD\_R\_SPEC''(m) \\ & \square\ sender\_dies \rightarrow SD\_R\_SPEC \end{aligned} \tag{10}$$

$SD\_R\_SPEC''(m)$ (Equation 11), outputs the data-message m to the receiver backend, by outputting it on channel *out*. Upon successfully outputting the data-message the process transits to SD_R_SPEC, this completes an uninterrupted message transfer. If instead *sender_dies* occurs the receiver still outputs the previously received data-message. This is modelled by $SD\_R\_SPEC''(m)$ recursing upon engaging in *sender_dies*.

$$\begin{aligned} SD\_R\_SPEC''(m) = {} & out!m \rightarrow SD\_R\_SPEC \\ & \square\ sender\_dies \rightarrow SD\_R\_SPEC''(m) \end{aligned} \tag{11}$$

This completes the specification of how the RRABP system should behave to the outside world, when an unstable sender is in use.

2.2.3. Properties of SD_SPEC

The process *SD_SPEC* (Equation 4) is expected to be non-deterministic, deadlock, and livelock free. The specification *SD_SPEC* is non-deterministic because the events *ready* and *tc* are hidden, *i.e.* internal. Thus the outside cannot determine whether or not the input by *SD_S_SPEC* has been transferred to *SD_R_SPEC*, before the sender has been replaced. To check whether or not this is the case we employed FDR2, Figure 5 shows that FDR supports our expectations. This completes the specification of the protocol.

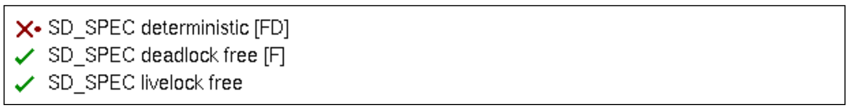
```
✗• SD_SPEC deterministic [FD]
✓ SD_SPEC deadlock free [F]
✓ SD_SPEC livelock free
```

Figure 5. FDR output after checking determinism, deadlock and livelock properties of *SD_SPEC*.

3. Formal Protocol Model

The RRABP deals with the situation that the sender, and *only* the sender, can be replaced at any moment. Replacing sender *S* in the system of Figure 1 (page 205), with a new instance of the sender *S*, means that receiver *R* and the unreliable bidirectional communication channel *CHAN* stay operational. Both *R* and *CHAN* have memory, this means that they can carry on with their normal operation using the messages currently available to them. The synchronisation routine, for the initial synchronisation, must reliably reset *R* independent of the state *R* or *CHAN* are in. Furthermore, it needs to deal with the properties of *CHAN*, which can lose or replicate data- and synchronisation-messages.

To overcome the degrading communication channel effects, the protocol applies the idea of the Alternating Bit Protocol. *S* repeatedly sends out copies of a message until it receives an acknowledgement from *R*. Because messages as well as acknowledgements can be lost by *CHAN*, *R* has to acknowledge every message it receives. To filter out replications during the synchronisation process, the protocol uses different messages instead of the tag-bit. The main problem, when designing the synchronisation part of the RRABP, was to find the correct number of synchronisation-messages.

3.1. Determining the Necessary Number of Synchronisation Messages

Initially, we thought one message would be sufficient. This however, is only the case if *CHAN* is replaced together with the sender, *i.e.* if *CHAN* gets replaced together with the sender *S*[1]. The problem that occurs with only a single reset-message is that the reset-acknowledgement may still be waiting for delivery in $C(c, d, 4)$, while the receiver has already processed its first data-message and is about to acknowledge it. If *S* now gets replaced, the new sender instance assumes, after receiving a reset-acknowledgment from the previous receiver synchronisation, that *R* is ready for a data-message. Thus the sender will not generate a reset-message, but instead send a data-message with the tag-bit set to 0. The receiver however classifies this data-message as a replication of a previous data-message and therefore discards it. Forcing the sender to always generate at least one reset-message, is not a solution, because this message may be lost by the channel. Finally, this implementation also violates the specification *SD_SPEC* in another respect. The new instance of *S* can input new data before *R* has output the data sent by the previous instance of *S*[2].

It is possible to construct a similar example when using two synchronisation-messages, which does not behave correctly when *S* gets replaced twice within a very short interval[3]. So we increased the number of synchronisation-messages to three for the RRABP, and we have used FDR to demonstrate that this solved the above problem. The remainder of this section explains how the protocol works. The three synchronisation messages used in the RRABP are: *stop*, *reset*, and *start*. The next section details their meaning and encoding together with the remainder of the protocol vocabulary.

3.2. Protocol Vocabulary

The RRABP uses the following messages:

1. *stop.msg*: Informs the receiver that the sender has just been started and is now synchronising with the receiver.

[1]The corresponding CSP$_M$ script is available at:
`http://www.abdn.ac.uk/piprg/Papers/CPA2008/RRABP/rrabp_sm_SC_R.csp`
[2]The corresponding CSP$_M$ script is available at:
`http://www.abdn.ac.uk/piprg/Papers/CPA2008/RRABP/rrabp_sm_S_RC.csp`
[3]The corresponding CSP$_M$ script is available at:
`http://www.abdn.ac.uk/piprg/Papers/CPA2008/RRABP/rrabp_dm_S_RC.csp`

2. *stop.ack*: Acknowledgement from the receiver that it received the stop message.
3. *reset.msg*: Informs the receiver that it should reset itself. This is the second message of the synchronisation sequence.
4. *reset.ack*: Reset acknowledgement from the receiver.
5. *start.msg*: Informs the receiver that the sender wants to start data communications. This is the third and last message of the synchronisation sequence.
6. *start.ack*: Acknowledgement from the receiver that it is ready to start data communication.
7. *data.msg.s.d*: This message transports the data *d* from the sender to the receiver, and *s* represents the tag-bit.
8. *data.ack.s*: This message acknowledges the correct handling of a data message with the tag-bit *s*.

3.2.1. Encoding the Vocabulary

In the description of the protocol we use the following data-types to construct the messages exchanged between sender and receiver:

- *OP* represents the different operations:

$$\textbf{datatype}\ OP = stop \mid reset \mid start \mid data \tag{12}$$

- *MT* represents the different message types:

$$\textbf{datatype}\ MT = msg \mid ack \tag{13}$$

- The set *TAG* represents the tag-bit:

$$TAG = \{0, 1\} \tag{14}$$

- The set *DATA* represents the data that can be transferred by the system:

$$DATA = \{0, 1\} \tag{15}$$

The type for all messages is the tuple: *OP.MT.TAG.DATA*. Furthermore, the encoded data-messages are defined as:

- *data.msg.s.d*: This message transports the data *d* from the sender to the receiver, with *s* representing the tag-bit.
- *data.ack.s.d*: This message acknowledges the correct handling of a data message with the tag-bit *s*. The data part *d* of the message must be ignored[4]!

For the synchronisation-messages: *stop.msg.s.d*, *stop.ack.s.d*, *reset.msg.s.d*, *reset.ack.s.d*, *start.msg.s.d*, and *start.ack.s.d*, the tag-bit *s* and the data part *d* are irrelevant.

3.3. Formal Model of Message Exchanges

This section details the communication between sender *S* and receiver *R*. It explains both the internals of sender and receiver.

[4]The inclusion of the data part is necessary because FDR insists on channel structure having the same number of components.

3.3.1. Sender

The entry point of the sender is $\mathit{SENDER}(\mathit{cin}, \mathit{cout})$ (Equation 16), the where *cin* is the channel from which the process reads the acknowledgements, and *cout* is the channel to which the process sends synchronisation- and data-messages. The $\mathit{SENDER}(...)$ process transfers stop-messages to the receiver. This is achieved by outputting the stop-message ($\mathit{stop.msg}.0.0$) on channel *cout* before recursing, until receiving a stop-acknowledgement ($\mathit{stop.ack.x.y}$) in reply. Upon which the process transits to $\mathit{S\_RESET}(\mathit{cin}, \mathit{cout})$ (Equation 17). Any other synchronisation- ($\mathit{reset.ack}, \mathit{start.ack}$) or data-acknowledgement ($\mathit{data.ack}$) pending on channel *cin* are swallowed by $\mathit{SENDER}(\ldots)$, to avoid deadlocks.

$$\begin{aligned}
&\mathit{SENDER}(\mathit{cin}, \mathit{cout}) = \\
&\quad \mathit{cout}!\mathit{stop.msg}.0.0 \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{stop.ack.x.y} \rightarrow \mathit{S\_RESET}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{reset.ack.x.y} \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{start.ack.x.y} \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{data.ack.x.y} \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout})
\end{aligned} \tag{16}$$

The $\mathit{S\_RESET}(\mathit{cin}, \mathit{cout})$ process is similar to the $\mathit{SENDER}(\ldots)$ process, except that it sends reset-messages to the receiver and expects a reset-acknowledgement before transiting to the state $\mathit{S\_START}(\mathit{cin}, \mathit{cout})$ (Equation 17). The process swallows any stop-acknowledgements, which are duplicates of a previously received stop-acknowledgement. However, at no time should the process receive a data-acknowledgement or a start-acknowledgement. Occurrences of these signal that the previously received stop-acknowledgement was not triggered by the sent stop-message. Therefore, the sender must try to reset the receiver once more. To do so the process transits to the state $\mathit{SENDER}(\ldots)$.

$$\begin{aligned}
&\mathit{S\_RESET}(\mathit{cin}, \mathit{cout}) = \\
&\quad \mathit{cout}!\mathit{reset.msg}.0.0 \rightarrow \mathit{S\_RESET}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{reset.ack.x.y} \rightarrow \mathit{S\_START}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{stop.ack.x.y} \rightarrow \mathit{S\_RESET}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{start.ack.x.y} \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout}) \\
&\quad \Box\ \mathit{cin}?\mathit{data.ack.x.y} \rightarrow \mathit{SENDER}(\mathit{cin}, \mathit{cout})
\end{aligned} \tag{17}$$

The last process, concerned with resetting the receiver is $\mathit{S\_START}(\mathit{cin}, \mathit{cout})$ (Equation 18). This process starts the receiver to make it ready to receive data-messages. This is done by outputting the start-message ($\mathit{start.msg}.0.0$) on the channel *cout* and then recursing, until a start-acknowledgement ($\mathit{start.ack.x.y}$) arrives on channel *cin*. Upon reception of the start acknowledgement the process transits to $\mathit{S\_RUN}(\mathit{cin}, \mathit{cout}, 0)$, the 0 represents the initial value of the tag-flag. Furthermore, the process filters any reset-acknowledgements. Reception of a data-acknowledgment means that the receiver is completely out of sync, hence it is necessary to restart the reset sequence again. So the process transits to $\mathit{SENDER}(\ldots)$. Receiving of a stop-acknowledgement results in a transit to $\mathit{S\_RESET}(\ldots)$, because this represents the state the receiver is currently in.

$$\begin{aligned}
&S\_START(cin, cout) = \\
&\quad cout!start.msg.0.0 \rightarrow S\_START(cin, cout) \\
&\quad \Box\ cin?start.ack.x.y \rightarrow S\_RUN(cin, cout, 0) \\
&\quad \Box\ cin?reset.ack.x.y \rightarrow S\_START(cin, cout) \\
&\quad \Box\ cin?stop.ack.x.y \rightarrow S\_RESET(cin, cout) \\
&\quad \Box\ cin?data.ack.x.y \rightarrow SENDER(cin, cout)
\end{aligned} \tag{18}$$

The main task of the process $S\_RUN(...)$ (Equation 19) is to wait for a message (m) to be transmitted from its backend over the channel in. Once it has received m the process transits to $S\_RUN'(\ldots)$ (Equation 20). The process swallows any reset-, start-, and data-acknowledgments, to prevent deadlocks.

$$\begin{aligned}
&S\_RUN(cin, cout, s) = \\
&\quad in?m \rightarrow S\_RUN'(cin, cout, s, m) \\
&\quad \Box\ cin?reset.ack.x.y \rightarrow S\_RUN(cin, cout, s) \\
&\quad \Box\ cin?start.ack.x.y \rightarrow S\_RUN(cin, cout, s) \\
&\quad \Box\ cin?data.ack.x.y \rightarrow S\_RUN(cin, cout, s)
\end{aligned} \tag{19}$$

$S\_RUN'(cin, cout, s, m)$ (Equation 20) appends the tag-bit to the message m and sends this data-message to the receiver before recursing. The process waits for the corresponding data-acknowledgement upon which it toggles the tag-bit and transits to $S\_RUN(\ldots)$. The process filters data-acknowledgements for a previous data-message as well as any reset- and start-acknowledgements.

$$\begin{aligned}
&S\_RUN'(cin, cout, s, m) = \\
&\quad cout!data.msg.s.m \rightarrow S\_RUN'(cin, cout, s, m) \\
&\quad \Box\ cin?data.ack.s.y \rightarrow S\_RUN(cin, cout, 1 - s) \\
&\quad \Box\ cin?data.ack.(1 - s).y \rightarrow S\_RUN'(cin, cout, s, m) \\
&\quad \Box\ cin?reset.ack.x.y \rightarrow S\_RUN'(cin, cout, s, m) \\
&\quad \Box\ cin?start.ack.x.y \rightarrow S\_RUN'(cin, cout, s, m)
\end{aligned} \tag{20}$$

This completes the sender model, now follows the description of the receiver model.

3.3.2. Receiver

The receiver consists of five different processes: $R\_RUN(\ldots)$, $R\_RUN'(\ldots)$, $R\_STOP(\ldots)$, $R\_START(\ldots)$, and $RECEIVER(\ldots)$. During normal operation the receiver toggles between $R\_RUN(\ldots)$ and $R\_RUN'(\ldots)$. The process $RECEIVER(\ldots)$ represents the entry point of the receiver, it determines the current value of the tag-bit. The remaining two processes synchronise the value of the tag-bit when the sender gets replaced.

Initially, the receiver side data handling functionality is in the state $RECEIVER(\ldots)$ (Equation 21). In this state it waits for a data-message to arrive: $cin?data.msg.x.m$, where the variable x holds the value of the tag-bit, and the variable m the received message. After receiving a data-message the process outputs the message (m) to the backend and sends a data-acknowledgment to the sender. After this the receiver transits to the state $R\_RUN(cin, cout, s, m)$ (Equation 22), setting the parameter s to the value of $(1 - x)$ (toggling the tag-bit). This initial acceptance of any data-message allows an unstable receiver to synchronise with the sender upon a restart. However, there is the danger of duplicating the last data-message received before the replacement of the receiver. Due to accepting any arriving data-message as valid, there is no need to synchronise the tag-bit. Therefore, the $RECEIVER(\ldots)$ acknowledges any arriving stop-, reset-, or start-message and then recurses.

$$\begin{aligned}&RECEIVER(cin, cout) = \\ &\quad cin?data.msg.x.m \rightarrow cout!data.ack.x.0 \rightarrow R\_RUN(cin, cout, 1-x) \\ &\quad \Box\ cin?stop.msg.x.y \rightarrow cout!stop.ack.0.0 \rightarrow R\_RESET(cin, cout) \\ &\quad \Box\ cin?reset.msg.x.y \rightarrow cout!reset.ack.0.0 \rightarrow RECEIVER(cin, cout) \\ &\quad \Box\ cin?start.msg.x.y \rightarrow cout!start.ack.0.0 \rightarrow RECEIVER(cin, cout)\end{aligned} \tag{21}$$

$R\_RUN(cin, cout, s)$ (Equation 22) is similar to the previously discussed process $RECEIVER(\ldots)$, except that it has the ability to filter out replications of data-messages. The parameter s represents value of the tag-bit for new data-messages. Upon inputting a new data-message ($cin?data.msg.s.m$) the process outputs the received message (m) to the backend. Once this has happened, a data acknowledgement gets sent to the sender and the process toggles the tag-bit while recursing. After inputting a replication of a previous data-message ($cin?data.msg.(1-s).m$), the process acknowledges it and recurses. Upon receiving a stop message the process outputs a stop-acknowledgement on channel $cout$ then it transits to $R\_RESET(\ldots)$ (Equation 23). The process acknowledges any arriving reset- or start-messages and then recurses.

$$\begin{aligned}&R\_RUN(cin, cout, s) = \\ &\quad cin?data.msg.s.m \rightarrow cout!data.ack.s.0 \rightarrow R\_RUN(cin, cout, 1-s) \\ &\quad \Box\ cin?data.msg.(1-s).m \rightarrow cout!data.ack.(1-s).0 \rightarrow R\_RUN(cin, cout, s, m) \\ &\quad \Box\ cin?stop.msg.x.y \rightarrow cout!stop.ack.0.0 \rightarrow R\_RESET(cin, cout) \\ &\quad \Box\ cin?reset.msg.x.y \rightarrow cout!reset.ack.0.0 \rightarrow R\_RUN(cin, cout, s) \\ &\quad \Box\ cin?start.msg.x.y \rightarrow cout!start.ack.0.0 \rightarrow R\_RUN(cin, cout, s)\end{aligned} \tag{22}$$

$R\_RESET(cin, cout)$ (Equation 23) represents the first tag-bit synchronisation state of the receiver. The receiver enters this state when it receives a stop-message during normal operation conditions. In this state the receiver waits for a reset-message to arrive, which the receiver acknowledges and changes to $R\_START(cin, cout)$ (Equation 24). $R\_RESET(\ldots)$ acknowledges any incoming stop- and start-messages and then recurses.

$$\begin{aligned}&R\_RESET(cin, cout) = \\ &\quad cin?reset.msg.x.y \rightarrow cout!reset.ack.x.y \rightarrow R\_START(cin, cout) \\ &\quad \Box\ cin?stop.msg.x.y \rightarrow cout!stop.ack.0.0 \rightarrow R\_RESET(cin, cout) \\ &\quad \Box\ cin?start.msg.x.y \rightarrow cout!start.ack.0.0 \rightarrow R\_RESET(cin, cout)\end{aligned} \tag{23}$$

The receiver process $R\_START(cin, cout)$ (Equation 24) is the second state of tag-bit synchronisation. In this state the receiver expects to receive a start-message, which it acknowledges. The tag-bit synchronisation sequence is now complete and the tag-bit is now set to 0. Now the receiver transits to $R\_RUN'(cin, cout, 0)$, with the tag-bit set to 0. In state $R\_START(\ldots)$ the receiver acknowledges any incoming stop- and reset-messages before recursing.

$$\begin{aligned} &\mathit{R\_START}(cin, cout) = \\ &\quad cin?start.msg.x.y \rightarrow cout!start.ack.0.0 \rightarrow \mathit{R\_RUN}'(cin, cout, 0) \\ &\quad \Box\ cin?reset.msg.x.y \rightarrow cout!reset.ack.0.0 \rightarrow \mathit{R\_START}(cin, cout) \\ &\quad \Box\ cin?start.msg.x.y \rightarrow cout!start.ack.0.0 \rightarrow \mathit{R\_START}(cin, cout) \end{aligned} \quad (24)$$

This completes the RRABP message exchange model. The next section details the formal model verification.

4. Formal Model Verification

The formal description of the RRABP does not automatically guarantee that the protocol possesses all desired properties. However, using a formal specification makes it possible to establish the protocol properties. This section shows the compliance of the protocol with the specifications for both: normal operation *COPY* (Section 2.1), and systems with unstable senders *SD_SPEC* (Section 2.2). For model checking the the model checker FDR2 [13] was used.

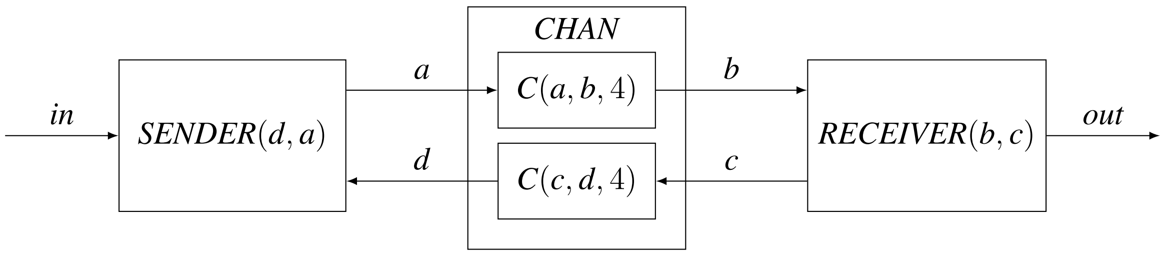

Figure 6. Process Network Layout for *RRABP_NO*.

4.1. Normal Operation

The process *RRABP_NO* (Equation 25) represents a communication system which consists of reliable sender and reliable receiver. They use the RRABP to communicate over an unreliable communication channel. The process, illustrated in Figure 6, is the parallel composition of *SENDER*(...) and *RECEIVER*(...) with *CHAN* (Equation 1), where *CHAN* represents the unreliable bidirectional communication channel.

$$\mathit{RRABP\_NO} = (\mathit{SENDER}(a, d) \underset{\{|a,d|\}}{\parallel} \mathit{CHAN}) \underset{\{|b,c|\}}{\parallel} \mathit{RECEIVER}(a, d) \setminus \{|\ a, b, c, d\ |\} \quad (25)$$

To prove that *RRABP_NO* is equivalent to *COPY* (Equation 3) we perform a failure divergence cross refinement check. Figure 7 shows that *RRABP_NO* has the same traces, failures, and divergences as *COPY*.

✓ COPY [FD= RRABP_NO
✓ RRABP_NO [FD= COPY

Figure 7. FDR screen-shot as proof that *RRABP_NO* is equivalent to *COPY*

This firmly establishes that the RRABP made the unreliable channel *CHAN*, reliable, as long as both sender and receiver stay operational.

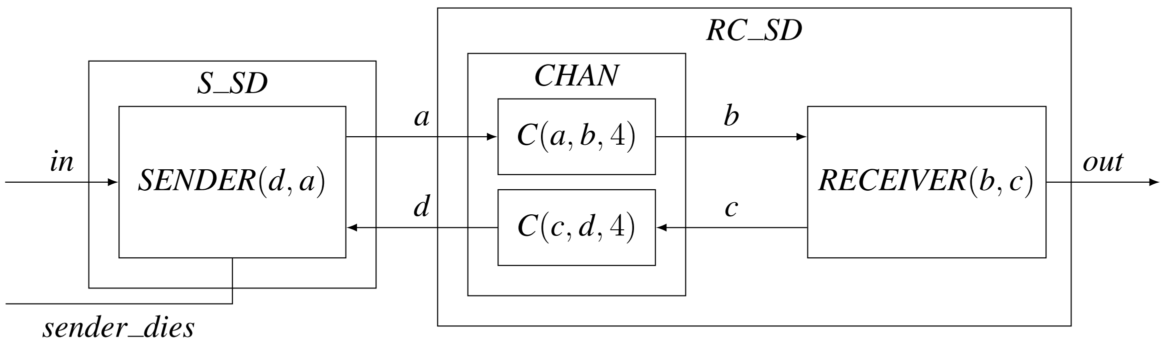

Figure 8. Process Network Layout for *RRABP_SD*.

4.2. Unreliable Sender

The process *RRABP_SD* (Equation 26) represents a communication system with an unreliable sender communicating with a receiver over an unreliable bidirectional communication channel. The sender and receiver of this system communicate using the RRABP. This system must comply with the specification given by *SD_SPEC* (Equation 4). Figure 8 illustrates the process network which represents the *RRABP_SD* process.

$$RRABP\_SD = S\_SD \underset{\{|a,d|\}}{\parallel} RC\_SD \setminus \{|\ a,b,c,d\ |\} \tag{26}$$

SD_RRABP is a parallel composition of two processes:

- *S_SD* (Equation 27) represents the sender of the communication system. The functionality covers unannounced sender replacement at any time. To model sender destruction it can be interrupted at any time by the event *sender_dies*. When this happens *SENDER*(...) passes control back to *S_SD*, which recursively creates a new instance of the sender.

$$S\_SD = SENDER(d,a) \triangle sender\_dies \rightarrow SD\_S \tag{27}$$

- *RC_SD* (Equation 28) represents the receiver and the communication channels of the system.

$$RC\_SD = RECEIVER(b,c) \underset{\{|b,c|\}}{\parallel} CHAN \setminus \{|\ b,c\ |\} \tag{28}$$

To check that the RRABP can deal with an unexpected sender restart, it is necessary to establish that *RRABP_SD* and *SD_SPEC* (Equation 4 on page 207) are equivalent. We establish this equivalence by performing a failure divergence cross refinement check of *SD_SPEC* and *RRABP_SD*. This refinement checks that both processes exhibit the same traces and fail (deadlock) or diverge (livelock) identically. Figure 9 shows that *RRABP_SD* has the same failures and divergences as *SD_SPEC*. This establishes security, stability, and functionality of the protocol even when a sender gets replaced during runtime. The complete model of the RRABP is available as CSP$_M$ script for download at: `http://www.abdn.ac.uk/piprg/Papers/CPA2008/RRABP/rrabp_tm_S_RC.csp`.

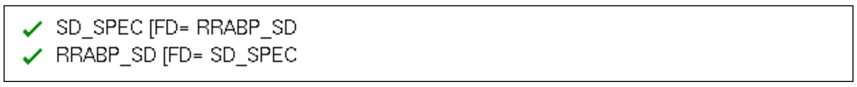
✓ SD_SPEC [FD= RRABP_SD
✓ RRABP_SD [FD= SD_SPEC

Figure 9. Proof of *RRABP_SD* is equivalent to *SD_SPEC*

5. Conclusion and Further Work

In this paper we establish that the RRABP enables reliable communication over an unreliable channel, which connects an unstable sender with a stable receiver. We underpin this claim by explaining how to model a point-to-point communication system using CSP, including a model of unreliable communication channels. This was followed by a description of the Alternating Bit Protocol, which forms the basis for the Resettable Receiver Alternating Bit Protocol (RRABP). After this, we developed the formal specification of the service the RRABP provides. This represents the specification of the RRABP. The RRABP was then outlined, first verbally and then formally. Finally, this formal model of the RRABP was proven to be equivalent to the previous given specifications.

During the development of the formal protocol description we discovered that a single reset message is not sufficient, if the communication channels stay operational while the sender is exchanged. In fact it was necessary to use a total of three different messages to trigger a reset of the receiver, fewer messages resulted in lost messages.

The RRABP is applicable in any environment where one wants to establish a reliable link with blocking capability over an unreliable (message replication and message loss) communication channel, which does not block. Furthermore, the protocol is applicable in environments with unstable senders. Initially, we targeted the RRABP at the water monitoring system of the WARMER consortium. In this system, unstable in-situ monitoring stations want to transfer data reliably to the data centre. Other environments which require the RRABP features include software defined radio systems which change their signal processing process networks over time, and sensor networks. Besides these systems, we are presently considering using the RRABP to establish a channel between a libCSP2 [14] process network executed by a PC and a libCSP2 process network executing on a soft processor within an FPGA on a PCI card. This requires extension of the RRABP to support bidirectional communication. Another interesting investigation is to determine the effect buffered communication channels have upon the RRABP. A last item, which is worth investigating, is to determine how the protocol copes with unstable senders and receivers.

Acknowledgements

This work has been supported by the EC FP6 project WARMER (**WA**ter **R**isk **M**anagement in **EuR**ope) project (contract no.: FP6-034472).

References

[1] F. Barnes and P. H. Welch. Communicating Mobile Processes. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[2] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.

[3] K. Chalmers and J. M. Kerridge. jcsp.mobile: A Package Enabling Mobile Processes and Channels. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter H. Welch, and David Wood, editors, *Communicating Process Architectures 2005*, pages 109–127, sep 2005.

[4] P. H. Welch. Graceful termination – graceful resetting. In Andrè W. P. Bakkers, editor, *OUG-10: Applying Transputer Based Parallel Machines*, pages 310–317, 1989.

[5] B. Sputh and A. R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter H. Welch, and David Wood, editors, *Communicating Process Architectures 2005*, September 2005.

[6] P. H. Welch, N. C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, July 2007.

[7] N. C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, sep 2004.

[8] G. H. Hilderink. Exception Handling Mechanism in Communicating Threads for Java. In Jan Broenink, Herman Roebbers, Johan Sunter, Peter H. Welch, and David Wood, editors, *Communicating Process Architectures 2005*, pages 317–334, sep 2005.

[9] B. Sputh, O. Faust, and A. R. Allen. A Versatile Hardware-Software Platform for In-Situ Monitoring Systems. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 299–312, July 2007.

[10] B. Sputh, O. Faust, and A. R. Allen. Integration of In-situ and remote sensing Data for Water Risk Management. In *Proceedings of the iEMSs* 4th *Biennial Meeting: "Integrating Sciences and Information Technology for Environmental Assessment and Decision Making"*, Burlington, USA, July 2008. International Environmental Modelling and Software Society. To be published.

[11] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–265, 1969.

[12] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, New Jersey 07485 United States of America, first edition, 1997. Download:
`http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf`
Last Accessed April 2008.

[13] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR Manual*. `http://www.fsel.com/fdr2_manual.html` Last Accessed June 2008.

[14] B. Sputh, O. Faust, and A. R. Allen. Portable CSP Based Design for Embedded Multi-Core Systems. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 123–134, September 2006.