BSP in CSP: Easy as ABC

Andrew C. Simpson, Jonathan M. D. Hill, and Stephen R. Donaldson

Oxford University Computing Laboratory, Oxford, United Kingdom

Abstract. In this paper we describe how the language of Communicating Sequential Processes (CSP) has been applied to the analysis of a transport layer protocol used in the implementation of the Bulk Synchronous Parallel model (BSP). The protocol is suited to the bulk transfer of data between a group of processes that communicate over an unreliable medium with fixed buffer capacities on both sender and receiver. This protocol is modelled using CSP, and verified using the refinement checker FDR2. This verification has been used to establish that the protocol is free from the potential for both deadlock and livelock, and also that it is fault-tolerant.

1 Introduction

In this paper we describe how the formal notation Communicating Sequential Processes (CSP) [6, 8] and the associated refinement checker FDR2 (for Failures-Divergences Refinement) [7] have been applied to the analysis of a transport layer protocol used in the implementation of the Bulk Synchronous Parallel model (BSP); the protocol is described in full in [2]. In particular, we describe how FDR2 has been used to establish that the protocol is free from the potential for both deadlock and livelock, and also that it is fault-tolerant.

The structure of this paper is as follows. In Section 2 we provide a brief introduction to BSP, while in Section 3 we describe the protocol with which we are concerned. In Section 4 we provide an introduction to the language of CSP, as well as the concept of refinement, which is fundamental to our formal analysis. Section 5 illustrates how CSP has been used to model the protocol, while in Section 6 we discuss how formal verification has been carried out using FDR2. Finally, Section 7 discusses the key issues raised in the paper, and provides some suitable conclusions.

2 The BSP Programming Model and *BSPlib*

BSP [11, 12, 10] is a model of parallel computation in which the abstract machine is a collection of p processor-memory pairs, an interconnection network or switch through which the processors communicate packets in a point-to-point manner, and a mechanism by which the processors can barrier synchronise.

A BSP program consists of a sequence of parallel *supersteps*, which are made up of three ordered phases. The first phase involves simultaneous local computation in each processor, in which only values stored in the processor's memory are

used. The second phase involves communication actions among the processors, which causes the transfer of data. Finally, a barrier synchronisation waits for the completion of all of the communication actions, and then makes any data transferred visible in the local memories of the destination processors.

In the absence of special-purpose BSP programming languages and computers, the BSP programming style can be realised by providing BSP communication primitives as a library which may be called from C or Fortran. Such a library is *BSPlib* [5], which provides a small number of communication primitives for BSP programming in an SPMD (single program, multiple data) manner. Data transfers are initiated by calls to several possible routines, e.g., `bsp_put(i,x,y,s)`, where `x` is a pointer to a data structure to be communicated, `y` is a data structure on `i` where a copy of `x` will be written, and `s` is the number of bytes transferred (i.e., the size of `x`). Barriers are entered by calls to `bsp_sync`, which is of course blocking: each independent thread invokes it when it has completed its computation, and returns from the invocation when all other threads have also invoked it. Calls to `bsp_sync` also trigger pending communications, since the implementation chooses to preserve the non-overlapping of computation and communication implied by superstep semantics.

In the next section we give an informal description of the implementation of *BSPlib*, and motivate why the system does not suffer from deadlock. We describe a low-level messaging layer (with fixed buffering capacity) that provides reliable delivery of packets. We aim to prove that this layer *and its use by BSPlib* guarantee freedom from both deadlock and livelock.

3 The messaging layer used by *BSPlib*

BSPlib is implemented on top of a *messaging* layer that provides a certain degree of isolation from the transport layer of any particular machine. The important primitives supplied by this layer are for hand-shaking and initialisation of the computation on the various processors, mechanisms for orderly and unorderly shutdown, a non-blocking send primitive, a blocking receive primitive, and a probe to test for message presence. It is important to realise that this is not a general interface; in particular, the implementation of the messaging layer takes into account how it is used by the *BSPlib* implementation. This restriction in the interaction between the two layers guarantees that a user's BSP programs are deadlock- and livelock-free; the purpose of this paper is to verify this claim.

In modern message passing systems, deadlock can be avoided if non-blocking communications are used in preference to blocking synchronous sends. This absence of deadlock is intuitive, as the use of non-blocking sends removes the circularity of pairing send-receive actions that gives rise to deadlock when communication is synchronous. However, non-blocking communications only guarantee this property if the buffering capacity of the protocol stack exceeds the amount of data that is pushed into the stack. Therefore, non-blocking communication can quite easily fail or suffer from deadlock if, for example, two processes simultaneously push data into their protocol stacks in an attempt to send large

amounts of data to each other. The point at which this deadlock occurs depends upon the buffering capacities of both sender and receiver. To quote the MPI report [4]: "*...the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. Quality implementations of MPI should ensure that this happens only in 'pathological cases'.*"

The BSP messaging layer implements the semantics of supersteps in the following way:

1. Calls to `bsp_sync` begin the actual transfer of data implied by previous communications in the superstep.
2. A total exchange (i.e., every processor communicates with every other processor) takes place between all the processors, so that each processor can inform the others of the number and size of messages it plans to send to them.
3. Each processor interleaves the sending of outgoing packets with the reception of any incoming packets destined for that processor. The emphasis of the interleaving is placed on reception so that messages need not be queued up at the receiver for a long time, wasting buffer resources.
4. When each processor has received all of the messages it is expecting, it proceeds into the computation phase of the next superstep.

Freedom from deadlock is guaranteed due to the fact that if there is insufficient buffering capacity in the protocol stack when a message is communicated in stages 2 and 3, then the non-blocking send fails and the implementation of *BSPlib* will always receive any packets that arrive. This reclaims the buffers associated with the received packets, which can be re-used for sending data. If this condition were not guaranteed, then deadlock may occur. Given our CSP description of the protocol, this possibility can be automatically detected (by FDR2).

3.1 A lower level transport layer

Despite the fact that *BSPlib* is built using a messaging layer which guarantees the ordered delivery of packets, this messaging layer is implemented upon the UDP/IP transport layer which does not. Therefore, although protocols such as UDP/IP have a shallower protocol stack and have a greater potential for high performance, if user APIs such as *BSPlib* are implemented over UDP/IP, they must handle the explicit acknowledgement of data and error recovery themselves.

To implement a reliable non-blocking send primitive, the implementation of *BSPlib* copies a user data structure—referenced in the send call—into a buffer that is taken from a free queue. If there are no free buffers available, then the send fails and it is up to the higher level *BSPlib* layer to recover from this situation. *BSPlib* recovers either by receiving any packets that have been queued for *BSPlib* or by asking those links that have a large number of unacknowledged send buffers to return an acknowledgement. The buffer is placed on a

send queue and a communication is attempted using the `sendto()` datagram primitive. Queueing the buffer on the send queue ensures that it does not matter if the packet communicated by `sendto()` fails to reach its destination, as our error recovery protocol can resend it if necessary. Only when a send has been positively acknowledged by the partner processor are buffers reclaimed from the send queue to the free queue.

Message delivery is asynchronous and accomplished by using a signal handler that is dispatched whenever messages arrive at a processor (SIGIO signals). Within the signal handler, messages are copied into user space by using `recvfrom()`.

We have used our own error recovery mechanism that uses a selective retransmission scheme that only resends the packets that are actually dropped and a sophisticated acknowledgement policy that is based upon available buffer resources and not timeouts [2]. The algorithm associated with the acknowledgement scheme has both a sender and receiver component. From the sender's perspective, if the number of buffers on a free queue of packets becomes low then outgoing packets are marked as requiring acknowledgements. This mechanism is triggered by calculating how many packets can be consumed (both incoming and outgoing) in the time taken for a message round-trip. The sender therefore anticipates when buffer resources will run out, and attempts to force an acknowledgement to be returned just before buffer starvation. This strategy minimises both the number of acknowledgements and the time a processor stalls trying to send messages. From the receiver's viewpoint, if n is the total number of send and receive buffers, then a receiver will only send a non-piggy-backed acknowledgement if it was requested and at least $\frac{n}{2p}$ messages have come in over a link since the last acknowledgement (either piggy-backed or explicitly).

3.2 A protocol for a reliable transport layer

Two types of packets are used in the protocol:

1. Sequenced data packets that contain a fixed-sized header with MAC, sequence number, payload length and type, and piggy-backed recovery information. This is followed by a variable-length payload, the length of which is bounded so that the entire packet is no larger than the physical layer's payload size (1500 bytes for Ethernet).
2. Unsequenced control packets that contain the same fixed-sized header, but no data. Control packets are used for explicit acknowledgements and *prods*, as described below.

Each processor contains a free queue of packets, and send and receive queues associated with each communication channel ($p-1$ in total). The send queue contains unacknowledged sent data-packets, and the receive queue contains data-packets not yet consumed by the upper *BSPlib* layer.

We describe a protocol which aims to minimise the number of redundant retransmissions of data packets and the number of control packets required to

achieve this. As an example, consider two processors—A and B—communicating over a channel that loosely guarantees packet ordering, but may drop packets. Suppose processor A sends the sequence of packets $\langle 0, 1, \ldots, 9\rangle$ (which will be queued on the send queue that A associates with B until acknowledged) to B, and only the sequence $\langle 0, 1, 4, 5, 6, 9\rangle$ arrives at B (and are queued onto a receive queue associated with A). If a go-back-n protocol (as used by TCP/IP) is used to recover from the lost packets, then processor A—after learning that the highest in-sequence packet received has sequence number 1—retransmits the packet sequence $\langle 2, 3, 4, 5, 6, 7, 8, 9\rangle$, which contains four redundant packets. A more sophisticated selective retransmit scheme would only resend the sequence $\langle 2, 3, 7, 8\rangle$. However, in both cases, it is the responsibility of the recovery protocol to ensure that the sending processor learns of these missing packets. This can either be done by piggy-backing recovery information on packets travelling in the reverse direction, or by sending an explicit control packet that informs the sender of the problem if there is no reverse traffic.

In our selective retransmission protocol, the fixed sized packet header contains two fields for piggy-backed sequence number information:

1. An *acknowledgement* field such that if B sent a message back to A after having received $\langle 0, 1, 4, 5, 6, 9\rangle$, this field would contain 1 (this allows A to remove all packets up to this value from the send queue).
2. An *end-of-hole* field which specifies the sequence number of the packet after the last in-sequence packet, i.e., 4. When no unaccounted-for packet loss has occurred, the acknowledged sequence number and this field are the same.

If A and B are involved in symmetric communication, then A will learn of the first hole from this piggy-backed data and resend the packets 2 and 3. After these have been received by B, further piggy-backed information will trigger the resending of 7 and 8. To prevent potentially stale in-flight information from being reused to cause repeated resends, a double retransmission of data is delayed for at least a round-trip delay of the network.

If there is no reverse traffic from B to A then if the upper *BSPlib* layer has consumed packets 0 and 1, and requires packet 2, B will *prod* processor A with a control packet. This allows A to learn of the hole and resend packets 2 and 3. In case the control packet goes missing, this prodding is repeated no more frequently than the round trip interval of the network. This prodding can also arise without packet loss when the computation on a processor runs ahead of another and hence enters its communication phase earlier. In this case, the prodded processor responds with an acknowledgement rather than missing data. The prodding processor then realises that it has run ahead and is being impatient, and enters an exponential back-off of prods.

There can be a problem with the protocol described above if the tail of a sequence of packets has been dropped. For example, consider if B only received the packets $\langle 0, 1\rangle$ and packets $\langle 2, 3, \ldots, 9\rangle$ have been dropped. Then, after the upper layer consumes the first two packets, it will send a prodding control packet to A with an acknowledgement and end-of-hole number of 1. When A receives this control packet, packets $\langle 0, 1\rangle$ will be acknowledged and removed from the

send queue, but—as there are more packets on the send queue—A will assume that they must have been dropped. Instead of just resending them all (which amounts to a go-back-n protocol), A resends the first and last packets from the send queue (i.e., 3 and 9). If there was no real hole then only two redundant messages are sent. On the other hand, if there really is a hole, then the receiver (eventually) gets an out-of-sequence message and can report the extent of the hole the next time the sender is prodded.

The benefits of this scheme over a more general selective retransmission scheme that could, for example, enumerate all the sequence numbers of packets dropped, is that only one integer field is required in the packet header. In the worst case, as many prodding control packets are generated as holes in the data; in the best, none. The scheme relies upon the observed fact that dropped packets will be clustered into holes. The network used, and the scheme we employ, ensures that packet loss is quite rare (0.05% packet drops are experienced in the NAS benchmarks [3]) and any packets lost are actually dropped by the receivers (for example, when available buffers run low). When buffers run low, only the next expected packets are accepted, but the acknowledgement data on all arriving packets is still inspected so that buffers held in the send queues can be freed and progress is guaranteed. This is proved in the remainder of the paper.

4 Communicating Sequential Processes

4.1 Events and processes

The language of CSP is a notation for describing the behaviour of concurrently-evolving objects, or *processes*, in terms of their interaction with their environment. This interaction is modelled in terms of *events*: abstract, instantaneous, synchronisations that may be shared between several processes.

We use compound events to represent communication. The event name $c.x$ may represent the communication of a value x on a *channel* named c. At the event level, no distinction is made between *input* and *output*. The willingness to engage in a variety of similar events—the readiness to accept input—is modelled at the process level. The same is true of output, which corresponds to an insistence upon a particular event from a range of possibilities.

A *process* describes the pattern of availability of certain events. The simplest process of all is *Stop*, which makes no events available at any time. The prefix process $a \rightarrow P$ is ready to engage in event a; should this event occur, the subsequent behaviour is that of P, which must itself be a process.

An external choice of processes $P \Box Q$ is resolved through interaction with the environment: the first event to occur will determine the subsequent behaviour. If this event was possible for only one of the two alternatives, then the choice will go on to behave as that process. If it was possible for both, then the choice becomes internal.

An internal choice of processes $P \sqcap Q$ will behave either as P or as Q. The result of this choice can be neither influenced nor predicted by the environment.

Both forms of choice exist in an indexed form: for example, $\sqcap i : I \bullet P(i)$ is an internal choice between processes $P(i)$, where i ranges over the (finite) indexing set I, while $\Box i : I \bullet P(i)$ is the corresponding external choice.

We may denote input in one of two ways. The process $c?x \rightarrow P$ is willing initially to accept any value (of the appropriate type) on channel c. On the other hand, if we wish to restrict the set of possible input values to a subset of the type associated with the channel c, then we may write $\Box x : X \bullet c.x \rightarrow P$. Furthermore, the processes $c?x?y \rightarrow P$ and $\Box x : X \bullet \Box y : Y \bullet c.x.y \rightarrow P$ are equivalent (assuming that X and Y the appropriate types).

In the parallel combination $P \parallel Q$, the component processes cooperate upon shared events. To identify which events are shared and which may be performed independently, we associate each component with an interface, or *alphabet*. If αP is the alphabet of P, then P will share in every event from this set; conversely, no event from αP can take place without P's participation.

We will sometimes wish to take the other extreme and insist that no events are shared, whatever the specified alphabets. We write $P \mid\mid\mid Q$ to represent the *interleaved* parallel combination of P and Q.

Finally, we may remove events from the interface of a process using the hiding operator. The process $P \setminus A$ behaves exactly as P except that events from the set A no longer require the cooperation of the environment; they are no longer visible to other processes.

4.2 Refinement

The standard notion of refinement for CSP processes, which is defined in [1], is based upon the failures-divergences model of CSP. In this model, each process is associated with a set of behaviours: tuples of sequences and sets that record the occurrence and availability of events.

The *traces* of a process P, given by *traces* $[\![P]\!]$, are finite sequences of events in which that process may participate *in that order*; the *failures* of P, given by *failures* $[\![P]\!]$, are pairs of the form (tr, X), such that tr is a trace of P and X is a set of events which may be refused by P after tr has occurred; and the *divergences* of P, given by *divergences* $[\![P]\!]$, are those traces of P after which an infinite sequence of internal events may be performed.

A process, Q, is said to be a *failures-divergences refinement* of another process, P, written $P \sqsubseteq Q$, if *failures* $[\![Q]\!] \subseteq$ *failures* $[\![P]\!]$ and *divergences* $[\![Q]\!] \subseteq$ *divergences* $[\![P]\!]$.

We may use this theory of refinement to investigate whether a potential design meets its specification. The refinement checker FDR2 [7] does exactly this. A pleasing feature of FDR2 is that if a potential design fails to meet its specification, a counter-example is returned to indicate why this is so.

5 A Formal Description

In this section we illustrate how the protocol described in Section 3 has been specified using the language of CSP.

5.1 Basic types and functions

We assume given types, *Node* and *Message*, which represent the processors associated with the protocol and the set of all messages respectively:

$$[Node, Message]$$

In addition, the type *Index* provides us with a means of associating sequence numbers with messages:

$$Index == \mathbb{N}$$

We assume the existence of an injective function, *number*, which associates sequence numbers with messages and functions, *sender* and *receiver*, which map messages to their sender and receiver respectively:

$$\begin{array}{l} number : Message \rightarrowtail Index \\ sender : Message \twoheadrightarrow Node \\ receiver : Message \twoheadrightarrow Node \end{array}$$

Finally, the free types *Probe_message* and *Probe_result* are concerned with communications associated with 'probes':

$$\begin{array}{rl} Probe\_message & ::= prod \mid back\_off \\ Probe\_result & ::= yes \mid no \end{array}$$

The former represent communications between processors checking for the availability of messages, while the latter represent similar communications between the upper and lower layer at given processors.

5.2 Events and channels

Prior to the protocol starting to transfer messages it is necessary for each node to establish exactly which messages it is to send. Thus, a communication of the form $to\_go.n_1.x$ informs node n_1 that x contains the messages which are to be sent (from n_1) and $to\_get.n_1.y$ indicates that y relates to the numbers of the messages which n_1 can expect to receive from other nodes. The event *synch* represents synchronisation between all of the processors after such communication.

The channels *send* and *receive* allow the processors to send and receive messages. As an example, the communication $send.n_1.n_2.m.2.4$ relates to node n_1 sending some message $m \in Message$ to node n_2, with 2 and 4 being the numbers of the last message acknowledged and number of the first message received 'after the window' respectively by n_1 from n_2. The communication $receive.n_2.n_1.m.2.4$ relates to the corresponding receipt.

The channel *send_prompt* models the action of one processor 'prodding' another for data, or replying to an earlier prompt from another processor. If the receiving node (for example, n_2) has no data to send, then its reply might be

of the form $send\_prompt.n_1.n_2.back\_off.1.3$. The channel *receive_prompt* relates to the receipt of such a prompt.

At a higher level, the channel *send_to* initiates the sending of a message from one node to another: it corresponds to the upper layer instructing the lower layer to send a message; the channel *can_send* is used to indicate whether or not there is buffer space available for such an action. In addition, the channel *out* passes messages from the lower layer to the upper layer.

Finally, communications on the channel *timeout* indicate that a timeout (due to non-receipt messages) has occurred at a particular processor. Such timeouts may only occur after an upper layer has sent all of its outgoing messages.

5.3 The upper layer

The upper layer is responsible for keeping track of the messages to be sent to, and to be received, from other nodes. Both kinds of information are represented by partial functions:

$$\begin{array}{|l} to : Node \pfun \text{seq } Message \\ from : Node \pfun \text{seq } Index \\ \hline this\_node \notin \text{dom } to \\ this\_node \notin \text{dom } from \end{array}$$

Initially, each node informs each other of the sequence numbers of the messages which it intends sending. We are not concerned with the explicit modelling of this phase here.

$$Upper(i) = to\_go.i?to \rightarrow to\_get.i?from \rightarrow synch \rightarrow Main(i, to, from)$$

After this initial phase, the behaviour of the upper layer depends upon the state of the system with regard to which messages have been received and sent.

$$\begin{array}{l} Main(i, to, from) = \\ \quad \textbf{if} \quad \forall x : \text{dom } to \bullet to\, x \neq \langle\rangle \wedge \forall x : \text{dom } from \bullet from\, x \neq \langle\rangle \\ \quad \textbf{then } Probe(i, to, from) \\ \quad \textbf{else if} \quad \forall x : \text{dom } to \bullet to\, x \neq \langle\rangle \\ \qquad \textbf{then } All\_Out(i, to, from) \\ \qquad \textbf{else if} \quad \forall x : \text{dom } from \bullet from\, x \neq \langle\rangle \\ \qquad\quad \textbf{then } Receive(i, to, from) \\ \qquad\quad \textbf{else } Upper(i) \end{array}$$

If there are still messages to be sent and to be received, then the upper protocol may try and probe; that is, it looks to see if there are any messages which are

ready to be passed from the lower layer:

$$
\begin{array}{l}
Probe(i, to, from) = Probe\_Yes(i, to, from) \\
\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\quad Probe\_No(i, to, from)
\end{array}
$$

$$
\begin{array}{l}
Probe\_Yes(i, to, from) = \\
\quad probe.i.yes \rightarrow out.i?m \rightarrow \\
\qquad \textbf{if} \quad number\, m = head\, (from\, sender(m)) \\
\qquad \textbf{then}\ Main(i, to, from \oplus \{sender(m) \mapsto tail(from\, sender(m))\}) \\
\qquad \textbf{else}\ Main(i, to, from)
\end{array}
$$

In the above the notation $f \oplus g$ indicates that the function f is *over-ridden* with the function g.

$$
\begin{array}{l}
Probe\_No(i, to, from) = \\
\quad probe.i.no \rightarrow \\
\qquad can\_send.i.yes \rightarrow \\
\qquad\quad \Box\, n : \{x : \mathrm{dom}\ to \mid to\, x \neq \langle\rangle\} \bullet \\
\qquad\qquad send\_to.i.n.head(to\, n) \rightarrow \\
\qquad\qquad\quad Main(i, to \oplus \{n \mapsto tail(to\, n)\}, from) \\
\qquad \Box \\
\qquad can\_send.i.no \rightarrow Main(i, to, from)
\end{array}
$$

If there is insufficient space in the buffer for a message to be passed from the upper layer to the lower layer, then the upper layer is forced to re-enter the *Main* process, which will check for incoming messages.

If the next message has arrived at the lower layer, then it is passed to the upper layer. On the other hand, if there is no message waiting to be passed from the lower layer then an outgoing message is passed to the lower layer.

If the node has received all of its messages but some are left to be sent, then the process *All_Out* is invoked:

$$
\begin{array}{l}
All\_Out(i, to, from) = \\
\quad \textbf{if} \quad \exists\, x : \mathrm{dom}\ to \bullet to\, x \neq \langle\rangle \\
\quad \textbf{then}\ can\_send.i.yes \rightarrow \\
\qquad\qquad \Box\, n : \{x : \mathrm{dom}\ to \mid to\, x \neq \langle\rangle\} \bullet \\
\qquad\qquad\quad send\_to.i.n.head(to\, n) \rightarrow \\
\qquad\qquad\qquad All\_Out(i, to \oplus \{n \mapsto tail(to\, n)\}, from) \\
\qquad\quad \Box \\
\qquad\quad can\_send.i.no \rightarrow Main(i, to, from) \\
\quad \textbf{else}\ Main(i, to, from)
\end{array}
$$

Here, the upper layer enters a phase where it continues to pass all messages which are left to be sent to the lower layer.

Finally, if all messages have been sent to the lower layer, but there are still messages to come in, then the upper layer enters a 'blocking receive' phase:

$$
\begin{array}{l}
Receive(i, to, from) = \\
\quad probe.i.yes \rightarrow out.i?m \rightarrow \\
\qquad \textbf{if} \quad number\, m = head\, (from\, sender(m)) \\
\qquad \textbf{then}\ Main(i, to, from \oplus \{sender(m) \mapsto tail(from\, sender(m))\}) \\
\qquad \textbf{else}\ Main(i, to, from) \\
\quad \Box \\
\quad probe.i.no \rightarrow Receive(i, to, from) \\
\quad \Box \\
\quad timeout.i \rightarrow Receive(i, to, from)
\end{array}
$$

This process keeps probing for data from the lower layer until messages are available to be output: it is only during this phase that a timeout can occur at the lower layer.

5.4 The protocol

At the lower level, the process *Protocol* is defined as follows.

$$
\begin{array}{l}
Protocol(i, msgs, last, ack, next) = \\
\quad Timeout(i, msgs, last, ack, next) \\
\quad \Box \\
\quad Receive(i, msgss, last, ack, next) \\
\quad \Box \\
\quad Send(i, msgs, last, ack, next) \\
\quad \Box \\
\quad Probed(i, msgs, last, ack, next) \\
\quad \Box \\
\quad Can\_Send(i, msgs, last, ack, next)
\end{array}
$$

In the above, the process *Protocol* is concerned with five parameters. The first, i, identifies the local processor, while *msgs* is the message buffer containing those messages which have been sent to, and received from, other nodes (and have not yet been acknowledged). The functions *last* and *ack* associate with each processor the sequence number of the last message seen (or the first message after a gap, if there is one) and the last message acknowledged respectively. Finally, the function *next* indicates the sequence numbers of the next expected message from each processor.

The process *Timeout* is defined thus:

$$
\begin{array}{l}
Timeout(i, msgs, last, ack, next) = \\
\quad timeout.i \rightarrow Send\_Prompts(i, msgs, last, ack, next)
\end{array}
$$

Here, if a timeout occurs then the other processors are prompted for data.

The process *Receive* is described thus:

$$
\begin{array}{l}
Receive(i, msgs, last, ack, next) = \\
\quad Receive\_Prod(i, msgs, last, ack, next) \\
\quad \Box \\
\quad Receive\_Back\_Off(i, msgs, last, ack, next) \\
\quad \Box \\
\quad Receive\_Message(i, msgs, last, ack, expecting)
\end{array}
$$

In the above, three communications may be received from another node: a 'prod', a 'back-off' warning, or a 'normal' message. The first of these cases—the receipt of a prod—is described by the following process:

$$
\begin{array}{l}
Receive\_Prod(i, msgs, last, ack, next) = \\
\quad receive\_prompt.i?n.prod?a?l \rightarrow \\
\qquad \textbf{if} \quad \neg\ \exists\, m : msgs \bullet receiver(m) = n \\
\qquad \textbf{then}\ send\_prompt.i.n.back\_off.ack(n).last(n) \rightarrow \\
\qquad\qquad\qquad Protocol(i, msgs, last, ack, next) \\
\qquad \textbf{else}\ \ Send\_All(i, msgs, last, ack, next, a, l, n)
\end{array}
$$

Here, if a prompt is received then either a 'back-off' warning is sent to the prodding node (if there are no missing messages) or all of the missing messages are sent to it; the process *Send_All* (which we do not describe here) takes care of the latter case. In addition to sending any missing messages, this process also updates the local state with respect to those messages which have been acknowledged by the sending processor.

If a 'back-off' message is received from another node then the message numbers of the last message received and the last acknowledgement sent are noted, and any missing messages are sent to that node:

$$
\begin{array}{l}
Receive\_Back\_Off(i, msgs, last, ack, next) = \\
\quad receive\_prompt.i?n.back\_off?a?l \rightarrow \\
\qquad Send\_All(i, msgs, last, ack, next, a, l, n)
\end{array}
$$

If a 'normal' message is received, then the process behaves thus:

$$
\begin{array}{l}
Receive\_Message(i, msgs, last, ack, next) = \\
\quad receive.i?n?message?a?l \rightarrow \\
\qquad \textbf{if} \quad \#\, msgs = capacity(i) - 1 \\
\qquad\qquad \wedge \\
\qquad\qquad next(sender(message)) \neq number(message) \\
\qquad \textbf{then}\ Send\_All(i, msgs, last, ack, next, a, l, n) \\
\qquad \textbf{else}\ \ Send\_All(i, msgs \cup \{message\}, last, ack, next, a, l, n)
\end{array}
$$

If the upper layer wishes to send a message then it is passed to the protocol which, in turn, passes it on:

$$
\begin{array}{l}
Send(i, msgs, last, ack, next) = \\
\quad send\_to.i?n?m \rightarrow send.i.n.m.ack(n).last(n) \rightarrow \\
\qquad Protocol(i, msgs \cup \{m\}, last, ack, next)
\end{array}
$$

The result of a probe by the upper layer is resolved as follows:

$$
\begin{array}{l}
Probed(i, msgs, last, ack, next) = \\
\textbf{if} \quad \exists\, n : Node \setminus \{i\};\ m : msgs \bullet next(n) = number(m) \\
\textbf{then}\ probe.i.yes \rightarrow \\
\qquad \Box\, n : \{n : Node \setminus \{i\} \mid \exists\, m : msgs \bullet next(n) = number(m)\} \bullet \\
\qquad out.i.(\mu\, m : msgs \mid number\ m = next(n)) \rightarrow \\
\qquad Protocol(i, \{m : msgs \mid sender(m) \neq n \vee number(m) \neq next(n)\}, \\
\qquad\qquad\qquad last, ack, next \oplus \{n \mapsto next(n) + 1\}) \\
\textbf{else}\ \ probe.i.no \rightarrow Protocol(i, msgs, last, ack, next)
\end{array}
$$

Finally, the ability for the upper layer to pass a message to the lower layer depends on the current state of the buffer:

$$
\begin{array}{l}
Can\_Send(i, msgs, last, ack, next) = \\
\qquad \textbf{if} \quad \#msgs < capacity(i) \\
\qquad \textbf{then}\ can\_send.i.yes \rightarrow Protocol(i, msgs, last, ack, next) \\
\qquad \textbf{else}\ \ can\_send.i.no \rightarrow Protocol(i, msgs, last, ack, next)
\end{array}
$$

5.5 The Protocol

For any processor $n \in Node$, we compose the upper and lower layers thus:

$$Node(n) = Protocol(n, \emptyset, last_n, ack_n, next_n) \parallel Upper(n)$$

We assume initial values of *last*, *ack* and *next* for each processor. We also assume the existence of a buffer for passing messages between pairs of nodes. For the moment, we assume such a buffer is perfect.

$$
\begin{array}{l}
Buffer(from, to) = \\
\qquad send.from.to?m?a?l \rightarrow \\
\qquad\qquad receive.to.from.m.a.l \rightarrow Buffer(from, to) \\
\qquad \Box \\
\qquad send\_prompt.from.to?p?a?l \rightarrow \\
\qquad\qquad receive\_prompt.to.from.p.a.l \rightarrow Buffer(from, to)
\end{array}
$$

The process *Comm* represents the interleaving of all such buffers:

$$Comm = \Big|\Big|\Big| i : Node;\ j : Node \mid i \neq j \bullet Buffer(i, j)$$

As such, our model of the protocol is given by the process *Comm_System*:

$$Comm\_System = (n : Node \bullet \Big|\Big| Node(n)) \parallel Comm$$

6 Formal Analysis

The refinement checker FDR2 (for Failures Divergences Refinement) [7] has been used in the analysis of our CSP description of the protocol (of which, the essential

elements have been described). This tool allows one to verify that potential implementations satisfy their specifications by checking that the former is a refinement of the latter.

We may attempt to establish the deadlock-freedom of *Comm_System* by testing whether it is a refinement of the process DF, where DF is defined by

$$DF = \bigsqcap e : \alpha\, Comm\_System \bullet e \rightarrow DF$$

We have established that two safety harnesses help ensure deadlock-freedom: these combine to ensure that there is always sufficient space in the lower layer's message buffer. The first condition ensures that messages can only be passed from the upper layer to the lower layer if there is sufficient space in the buffer (assuming that the upper layer behaves in accordance with messages received from the lower layer). The second condition ensures that if the buffer is reaching its capacity and an out-of-sequence message is received from another processor, then that message is thrown away (but the acknowledgement information is retained). If either of these conditions were to be removed, then the possibility of deadlock arises. (Strictly, of course, removing one (or both) of these conditions will not result in deadlock, but in *livelock*.)

Livelock and deadlock are related phenomena; indeed, the relationship between them is demonstrated by the means in which we apply FDR2 in the analysis of the possibility of livelock. In the context of our specification, we consider the protocol to be livelocked precisely when the lower layer embarks upon an infinite uninterrupted sequence of events, without any communication with the upper layer: from the point of perspective of the upper layer, the system is deadlocked.

If we let $\alpha\, Lower$ denote the set of all communications which the lower layer may observe and $\alpha\, Upper$ denotes the set of all communications which the upper layer may observe, we may test our protocol for livelock by investigating whether $Comm\_System \setminus \alpha\, Lower$ is deadlock-free. We have found this to be the case—assuming perfect communication.

In [9], a method for verifying the fault-tolerance of protocols using CSP and FDR2 is described. This approach involves ensuring that *critical* events are *protected* from potential faults or failures.

By replacing our *Buffer* processes with a process which represents faulty behaviour, we can verify whether our protocol is fault-tolerant. Such faulty buffers may nondeterministically create, drop, or resequence messages at any time. Provided that such behaviour is suitably bounded (i.e., there cannot be an uninterrupted, infinite sequence of dropped packets), we have been able to establish that our protocol is deadlock-free in the context of such faults.

If, however, either of our safety harnesses is removed, then it is no longer the case that our protocol is free from the potential for deadlock. For example, if we fail to ensure that only the next expected message can be received when there is limited space in the buffer, then it is possible for a series of out of sequence messages to arrive and fill up the buffer: this will result in deadlock.

7 Conclusions

We have shown that our protocol is fault-tolerant, and free from the potential for deadlock and livelock. As the protocol uses fixed buffer capacity, there is a potential for deadlock when buffers run low. Deadlock is avoided in this situation due to the manner in which packets are dropped on reception, but after their protocol information has been processed. Further, if buffers become low when sending a message, then the sender guarantees to back-off into the reception of a message. This protection mechanism restricts the ways in which the lower layer is used. It should be noted that this is not overly-restrictive, as *BSPlib* automatically guarantee this restriction. By contrast, an application using BSD stream sockets (i.e., TCP/IP) could easily suffer this fate without due care from the programmer (for example, by mismatching sends and receives, or by simultaneously sending too much data).

References

1. S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Proceedings of the NSF-SERC Seminar on Concurrency*, pages 281–305. Springer-Verlag Lecture Notes in Computer Science, volume 197, 1985.
2. S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. BSP clusters: high performance, reliable and very low cost. *Parallel Computing: Special Issue on Cluster Computing*, 1998. Submitted for publication.
3. S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. Performance results for a reliable low-latency cluster communication protocol. In *PC-NOW '99: International Workshop on Personal Computer based Networks Of Workstations, held in conjunction with IPPS'99 (under submission)*, 1999.
4. Message Passing Interface Forum. *MPI A Message-Passing Interface Standard*, May 1994.
5. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory, May 1997.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
7. A. W. Roscoe. Model checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall International, 1994.
8. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, 1997.
9. A. C. Simpson, J. C. P. Woodcock, and J. W. Davies. Safety through security. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 18–24. IEEE Computer Society Press, 1998.
10. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.
11. L. G. Valiant. A bridging model for parallel computation. *Journal of the ACM*, 33(8):103–111, August 1990.
12. L. G. Valiant. Bulk-synchronous parallel computer. U.S. Patent No. 5083265, 1992.