Softw Syst Model (2005) 4: 258–276 / Digital Object Identifier (DOI) 10.1007/s10270-005-0084-3

Investigating a file transfer protocol using CSP and B

Neil Evans[1], **Helen Treharne**[2]

[1] Department of Computer Science, Royal Holloway, University of London, UK
[2] Department of Computing, University of Surrey, UK

Published online: 11 May 2005 –

Abstract. In this paper a file transmission protocol specification is developed using the combination of two formal methods: CSP and B. The aim is to demonstrate that it is possible to integrate two well established formal methods whilst maintaining their individual advantages. We discuss how to compositionally verify the specification and ensure that it preserves some abstract properties. We also discuss how the structure of the specification follows a particular style which may be generally applicable when modelling other protocols using this combination.

Keywords: CSP – B – Combining formalisms – Compositional verification

1 Introduction

Large scale programming often adopts a strong separation between the model, view and controller aspects of a system [12]. One benefit of this widely used technique is the isolation of the state of the model from its co-ordinating software. This enables a programmer to focus on different parts of a system individually, which would be difficult to comprehend and maintain otherwise. We believe that a similar separation of concerns is also appropriate at the specification level so that we can concentrate on the state and co-ordinating aspects of a specification individually. State-based methods are primarily concerned with describing the state of a system together with its associated properties whereas eventbased methods support the formal description of components which interact in a co-ordinated way.

Our approach to the specification of complex interacting systems centres around the combination of an event-based method and a state-based method: CSP [14] and B [2]. Previous work [19, 20, 24, 25] has shown that this combination – which is called CSP ‖ B[1] – allows the two approaches to be integrated in a straightforward way. The integration relates CSP events with B operation calls, so that the original semantics of both languages is preserved. The main benefit of retaining the individual languages is that existing tool support for these formal methods is available immediately.

The overhead of the integration is minimal and involves extra consistency checks to ensure the state and event based descriptions are compatible. These consistency checks are akin to the establishment of design contracts between the individual components of a system. A key feature of our approach is that these checks can be carried out compositionally. This means that CSP ‖ B has the necessary foundations in order to be scalable and is appropriate to be used to specify systems that involve many interacting components.

In addition to consistency checking, the CSP ‖ B approach enables us to verify other properties of a system using the model checker FDR[2]. This tool is designed specifically for the analysis of CSP processes. However, theoretical results allow us to infer properties of an entire combined system (properties such as deadlock, livelock, and process refinement) by examining the event based descriptions of the system in isolation. Establishing these properties may require the introduction of further design contracts. During verification, we move between the state-based model and the event-based model of the system in order to introduce any additional design contracts systematically.

In this paper we present the formal investigation of a file transmission protocol to illustrate how we develop and verify properties using the CSP ‖ B combination. By using a standard example, the Bounded Retransmis-

[1] This is pronounced ‘CSP parallel B’.
[2] `http://www.fsel.com`

sion Protocol [13], we aim to emphasise the different features of CSP ‖ B, and show how existing tool support is utilised. However, in addition, we introduce a new, compositional approach to CSP ‖ B verification that makes larger specifications amenable to model checking techniques. The verification of the protocol under investigation will apply this compositional approach.

The protocol is unusual in that it contains complex control flow and also manipulates data in an interesting way. Describing the protocol using only a state-based method would involve the use of auxiliary variables to encode the information flow, and for complex control flow this can be cumbersome. Similarly, using only an event-based method to describe the protocol would mean that the clarity of the control flow is obscured by the state information that would be present in the description. Therefore, using a combination of a state and event-based method to specify the protocol seems entirely appropriate. We show that for the specification of the Bounded Retransmission Protocol we adopt a particular architectural style which clearly identifies the participating components and their interactions. This facilitates consistency checking and the automated verification process.

The rest of the paper is structured as follows: Sect. 2 provides the motivation for the combined approach, as well as the necessary technical background. Section 3 details the protocol itself, and Sect. 4 demonstrates how it is modelled using CSP ‖ B. This includes a new feature of the CSP ‖ B approach whereby a synchronisation between two CSP processes is required to perform a machine operation. A formal analysis of the model is given in Sects. 5 and 6. Previous work on the theory of CSP ‖ B is illustrated in Sect. 5. This work is put into context by highlighting similarities with the work of Lamport and Schneider in [15]. Section 6 demonstrates some new results through the formal analysis of the protocol. These results can be exploited by the approach in other application areas.

The protocol described in the paper has been investigated using other approaches including Event B [4], LOTOS [16], PVS [11], and I/O Automata [13]. Section 7 makes comparisons between our CSP ‖ B approach, the Event B approach [3] (which is a specialisation of the B method), and also the LOTOS approach [6] (which is a language based on CSP). Section 7 also summarises the overall results of the analysis and discusses their impact on the CSP ‖ B approach in general.

2 Background

As we stated above, one motivation for our CSP ‖ B approach is to make use of existing tool support to develop computer systems with both complex state and control (event based) requirements. These tools (i.e. FDR and the B Toolkit) have been used successfully in industrial strength projects [5, 23]. The aim of our combination is to improve the expressiveness of formal specifications whilst maintaining the advantages of each individual approach.

The model checker FDR provides highly automated tool support for the analysis of (event based) CSP processes. It is capable of determining deadlock and divergence (livelock) freedom of individual processes, and can perform checks for more abstract patterns of behaviour using CSP refinement for a variety of semantic models.

The B Toolkit[3] is an environment for the development of provably correct software. An abstract state based specification (comprised of modular units called *machines*) is successively refined by adding more and more detail until it is at a sufficiently low level that code can be generated automatically. Properties of the specification are expressed as an invariant of the state space. The refinement procedure of the B method ensures that more concrete versions maintain invariants that are at least as strong as the original.

2.1 CSP ‖ B

The fact that we can consider the combination of CSP processes and B machines is due primarily to Carroll Morgan's CSP semantics of action systems [17]. This enables us to give corresponding semantics to B machines.

The three most frequently used denotational semantics for CSP are the *traces* model, the *stable failures* model, and the *failures/ divergences* model (see [18, 21] for more details). A *trace* is a finite sequence of events that belong to a global set of events, Σ. A trace tr is said to be a trace of a process P if the process can perform the sequence of events in tr. In the traces model, a process P is identified with the set of traces that it can perform. A *stable failure* is a pair (tr, X) consisting of a trace tr and a set of events X (called a *refusal* set). This pair is said to be a stable failure of a process P if it can perform the trace tr and then, on the proviso that the process cannot perform any hidden events, it is unable to perform any event in X. In the stable failures model, a process is identified with the set of trace/refusal pairs that it exhibits together with its set of traces. A *divergence* is also a sequence of events. A trace tr is a divergence of a process P if it can perform an infinite succession of internal events after performing some prefix of tr. A *failure* is also a trace/refusal pair. A pair (tr, X) is a failure of a process P if tr is a divergence of P or (tr, X) is a stable failure of P. In the failures/divergences model, a process is identified with the set of failures that it exhibits together with its set of divergences.

For any B machine M, the sequence of operation calls $\langle e_1, e_2, \ldots, e_n \rangle$ is a *trace* of M if such a sequence is possible in M (i.e. it is not guaranteed to block). In AMN [2] this is written as the formula $\neg([e_1; e_2; \ldots; e_n]\mathit{false})$. Such a sequence is a *divergence* if it is not guaranteed to terminate (i.e. performing the corresponding sequence of operation calls does not establish the postcondition *true*);

[3] `http://www.b-core.com`

this is written as $\neg([e_1; e_2; \ldots ; e_n]true)$. In a B machine M, each operation e has a guard g_e. (If a guard is not defined explicitly, then it is considered to be *true*.) Given a set of events X, $(\langle e_1, e_2, \ldots , e_n\rangle, X)$ is a *failure* of M if $\neg([e_1; e_2; \ldots ; e_n](\bigvee_{e\in X} g_e))$ – i.e. none of the guards are true after performing the sequence of operation calls. Although Morgan does not give a stable failures definition for action systems, an intuitive definition is the set of trace/refusal pairs whose trace component is not a divergence. This is discussed in more detail in [20].

Now that a CSP semantics can be given to B machines, it is meaningful to speak of the parallel composition of a CSP process with a B machine; this is the reason for naming our approach CSP ‖ B. We shall be concerned with a particular architecture in which each B machine is composed with a unique CSP process called its *controller*; a B machine can only interact with the environment via its controller. A CSP ‖ B specification may consist of multiple controller/machine pairs. In this architecture, controllers can also interact with each other.

2.1.1 The controller language

We use a subset of CSP to define the controllers for B machines. This is essentially the sequential part of CSP with I/O communication over named channels, as is discussed in [19]:

$$\begin{aligned} P ::= {} & a \to P | c?x\langle E(x)\rangle \to P | d!v\{E(v)\} \to P | \\ & e!v?x\{E(x)\} \to P | P_1 \Box P_2 | P_1 \sqcap P_2 | \\ & \sqcap_{x|E(x)} P_x | \ \mathit{if}\, b\, \mathit{then}\, P_1\ \mathit{else}\, P_2\ \mathit{end}\, | S(p) \end{aligned}$$

Initially, when modelling a system, we use events such as a, $c?x$ and $d!v$ to allow controller processes to communicate with the external environment and interact with each other. The event names c and d are called *communication channels* because they allow the passage of data: the event $c?x$ inputs a value into a process by assigning the value to the variable x, and the event $d!v$ denotes the output[4] of the value v on channel d. In general, a communication channel is capable of passing both input and output data, and all values must be of the correct type. The event a is called a *synchronising event* because it does not pass data. Furthermore, we allow events of the form $e!v?x$, where e is called a *machine channel*. Machine channels enable a controller process to call operations within its corresponding B machine (this is discussed in more detail in Sect. 2.1.2).

Subsequently, during verification, we may annotate these variables and values with assertions to strengthen the CSP description. For example, the event $e!v?x\{E(x)\}$ (corresponding to a B operation) can accept any x as input, but will diverge if $E(x)$ is not satisfied. Hence, assertions of this kind are called *diverging assertions*. Similarly, the augmented event $e!v?x\langle E(x)\rangle$ will be blocked if $E(x)$ is not satisfied; these assertions are referred to as *blocking assertions*. As we shall see in Sect. 5, both kinds of assertion are needed to make design contracts explicit.

The other CSP operators used in the construction of the controller processes are external choice, (general) internal choice, and conditional choice. The expression $S(p)$ introduces recursion into process definitions. After construction, the controller/machine pairs are composed by using the parallel operator '‖' and, during the analysis phase, we also make use of the hiding operator '\', the sequential composition operator ';', and the interleaving operator '|||'.

2.1.2 Controller/machine synchronisation

Every machine channel in a controller process corresponds to an operation in a B machine. The structure of the events in the process must match the signature of the B operation. For example, the event $e!v?x$ in a controller process corresponds to a call $x \leftarrow e(v)$ of the operation e in the controller's B machine. Note that the output value, v, of the event becomes the input to the operation, and the event's input variable, x, is assigned the value output by the operation. The types of the values input and output by the operation must match the type of the channel e. In general, machine channels may also pass input values only, output values only or pass no values at all.

In Sect. 2.1, we stated that every B operation has a guard (that was defined explicitly, or defined implicitly as the predicate *true*). In the combined approach we restrict ourselves to operations without explicit guards – i.e. all guards are implicitly *true*. Such operations are referred to as *non-blocking* operations because they are enabled in every possible state, and for every input they must provide some output. (However, we allow preconditioned operations which, therefore, introduces the possibility of divergences in the B machine.) The reason for this constraint is because operations with non-trivial guards are not implementable in general.

We also require that all inputs on machine channels are *non-discriminating*. Intuitively, this means the CSP events corresponding to B operations cannot be selective with the inputs they are prepared to accept. More formally, if a controller can refuse an event $e.v.w$ for a particular input value w, then it must refuse all events of this form (regardless of the value w).

3 The Bounded Retransmission Protocol

The Bounded Retransmission Protocol (BRP) controls the transfer of data files over a communications medium. There are two entities involved in a run of the protocol: the *sender* and the *receiver*. They are separated by the *medium* over which messages can pass. The sender is will-

[4] In process definitions, the symbol '!' is merely syntactic sugar to indicate the output of a value. As such, we will often use the basic event structuring operator '.' in its place.

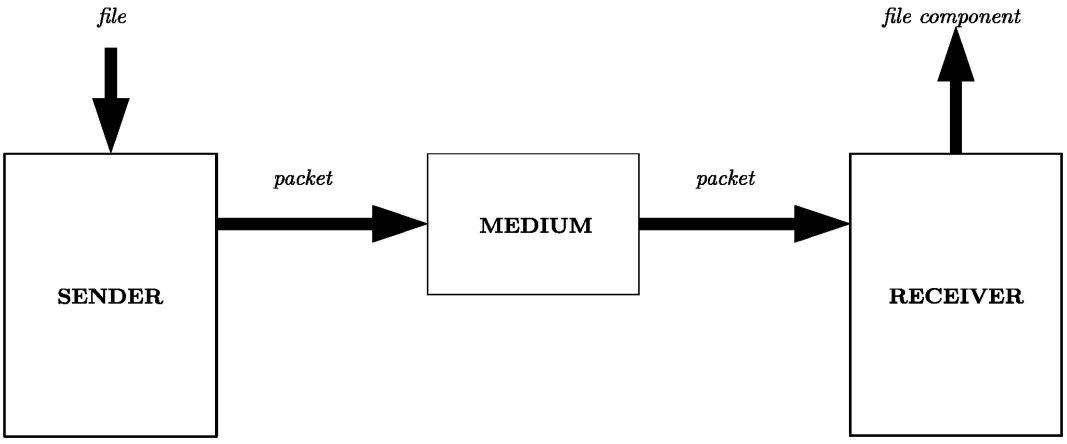

Fig. 1. The general architecture for the Bounded Retransmission Protocol

ing to accept an arbitrary sized file from its environment, and is responsible for splitting the file into fixed size file components to be transferred over the medium to the receiver. Each file component is sent as part of structured message called a *packet*. An entire file is sent to the receiver as a sequence of packets. On receipt of a packet, the receiver is able to extract the file component which it passes to its environment. The flow of information from the sender to the receiver is shown in Fig. 1.

The medium is unreliable because it has the potential to lose packets. The aim of the protocol is to overcome this deficiency by enabling the retransmission of packets that are lost in transit; a variation of the alternating bit protocol is used to achieve this. Thus, in addition to the file component, a packet contains a bit (called a *toggle*) whose role is to distinguish new packets from retransmissions. (In addition to the toggle, a packet contains two flags that indicate whether its file component is the first or last component of the file respectively.) The receiver replies to each successful packet transfer by sending an acknowledgement over the medium to the sender. It is also possible, therefore, for acknowledgements to be lost. The bounded nature of the protocol means that only a finite number of retransmissions are attempted before the sender aborts the run completely.

A confirmation signal is sent by the sender to the environment to indicate whether a file transfer has been successful (i.e. the system has reached a point in which all packets have been received and acknowledged), a possible success (i.e. the last packet has been sent without a subsequent acknowledgement), or aborted (i.e. after a bounded number of attempts to send an intermediate packet, the file transfer has been abandoned).

4 Modelling the protocol

From the description of the protocol given above, we begin to separate the various aspects of the protocol to construct a formal model. We adhere to the general structure depicted in Fig. 1, but now we add detail to the entities within the figure. We set up two controller/machine pairs that correspond to the sender and the receiver. The medium, whose role is simply to pass packets and acknowledgements (or lose them), has no complex state and, therefore, has no need for an underlying B machine.We also add channels between the various components in anticipation of the kinds of communications that will be needed. The resulting architecture is shown in Fig. 2.

SenderCtrl is the name given to the sender's controller process. The channel *req* allows files to pass from the external environment to the sender, and *conf* passes the confirmation signal from the sender to its environment. The channel *trans* allows packets to be passed from the sender to the medium, and *rec* allows their passage from the medium to the receiver's controller process (called *ReceiverCtrl*). On the receiver's side, the channel *ind* allows file components to be passed to the external environment, together with a label to indicate their relative position in the file (first, last or intermediate), and *ind_err* is a channel that fires whenever a file transfer has been aborted. Acknowledgements are passed from the receiver to the medium on the *sendack* channel, and *recack* allows their passage to the sender. The double arrows in the figure represent all machine channels; these are described in more detail below.

In the original protocol, retransmissions and, therefore, aborted runs are determined by timeouts: the sender starts a counter every time a packet is sent, and the receiver starts a counter every time an acknowledgement is sent. If, after a reasonable amount of time, the sender has not received an acknowledgement for the transmitted packet, a timeout occurs and the packet is retransmitted (until an abort is performed). Similarly, if the receiver fails to receive an expected packet within a certain period of time, a timeout occurs and the protocol is aborted.

Since the protocol relies on the correctness of the timeouts, and because FDR does not implement time, we model timeouts by adding extra synchronisation chan-

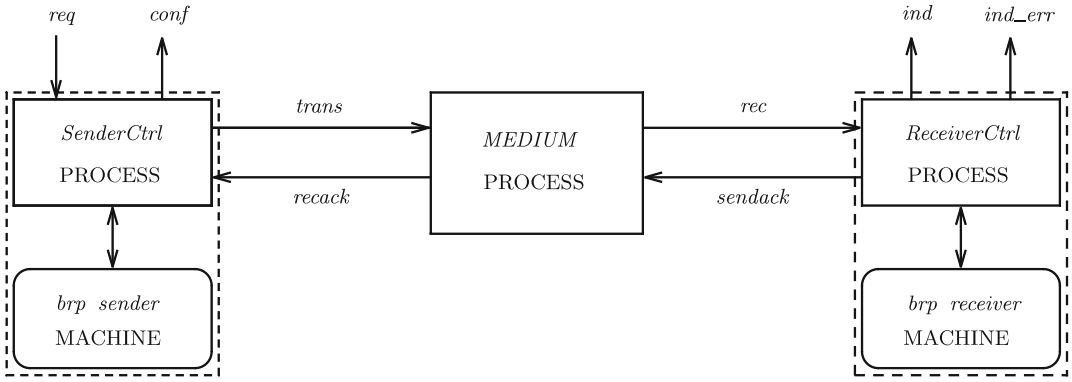

Fig. 2. The formal architecture for the Bounded Retransmission Protocol

nels; this is consistent with all of the approaches that have analysed the BRP [4, 11, 13, 16].We introduce a synchronisation channel between the sender and medium called *dec* that fires precisely when a packet or an acknowledgement is lost. This enables the modelling of timeouts on the sender side. Similarly, we introduce a synchronisation channel between the sender and the receiver called *abort* that forces them to abandon a run of the protocol at the same time. This is shown in Fig. 3. By modelling timeouts as synchronisations we avoid analysing situations where, for example, the receiver aborts a run while the sender proceeds with the protocol run. The protocol does not aim to cope with such pathological situations.

4.1 The sender side

In constructing the *SenderCtrl* process, we can identify two distinct patterns of behaviour: the first is when the sender is prepared to accept new files from its environment, and the second is when the sender is transferring packets to the receiver. As a state based requirement, it is the responsibility of the sender's B machine to hold the file components that have yet to be transmitted and construct the packets that are to be sent over the medium; the number of retransmissions that the sender is prepared to do before aborting is also retained in its B machine. In addition, the confirmation signal is (partially) dependent on the state of the file transfer, and we therefore conclude that the status of this signal should be (partially) resolved by the B machine. This motivates the following signatures for the (as yet undefined) operations of the sender's B machine:

- **initiate**(*file*) accepts a file as input and stores it as a piece of state.
- *ff, ll, tt, mm* ← **get_packet** outputs the current packet to be sent over the medium. The output parameters correspond to the four components of a packet: the first component flag, the last component flag, the toggle, and the file component.
- **advance** updates the state when an acknowledgement has been received so that the next packet is ready to be sent. This involves negating the toggle so that the packet can be distinguished from the last.
- *ss* ← **get_status** outputs the status of the file transfer for use in the confirmation signal.
- *nn* ← **retrans** outputs the number of retransmissions that the sender is prepared to do.
- **dec** decrements the number of retransmissions.

Note that *dec* was introduced above as a synchronisation channel to model timeouts. Therefore, a call of this operation requires the co-operation of the sender's control process and the medium. This is a departure from the constraints on the general architecture given in Sect. 2.1 where a B machine must be associated with a unique controller. We can informally justify this, however, by observing that the synchronisation will only restrict the calls of the operation. Thus, we cannot witness any extra behaviour (such as calling the operation outside its precondition) by imposing such a restriction. This is discussed in more detail in Sect. 7.

The machine channels listed above, together with the communication and synchronisation channels that have already been defined, motivate the mutually recursive process definition of the *SenderCtrl* process shown in Fig. 4. Note that a file is defined to be a sequence of file components, and the successful completion of a file transfer (signalled by *conf.ok*) can be determined purely within the process itself. Hence, if the file is empty then a *conf.ok* signal can be issued immediately. Also, on the successful completion of a non-empty file transfer, a **flip** operation is called prior to the confirmation signal. This operation is required to negate the toggle so that the first packet of a new file can be distinguished from the last packet of the current file.

The declaration of the sender machine's state variables is shown in Fig. 5. The variable *file_left* holds the file components that have yet to be transferred successfully. Since the completion of a file transfer can be determined within the controller process itself, the state variable *status* determines the confirmation signal for unsuccessful file transfers only. Thus, it is assigned a value *not_ok* when the protocol has aborted before the last

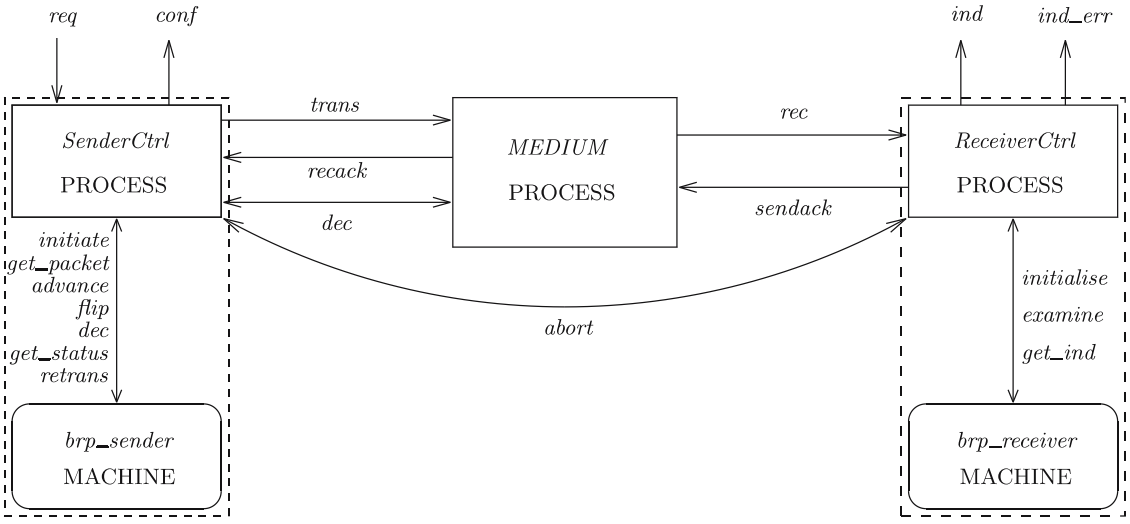

Fig. 3. All channels for the Bounded Retransmission Protocol

$$
\begin{array}{l}
SenderCtrl \;=\; \\
\quad req?file \rightarrow \mathit{if}\ file \;=\; <> \ \mathit{then}\ conf.ok \rightarrow SenderCtrl \\
\qquad\qquad\qquad\qquad \mathit{else}\ initiate.file \rightarrow SendPacket \\
SendPacket \;=\; \\
\quad retrans?n \rightarrow \\
\quad\quad \mathit{if}\ n \;>\; 0\ \mathit{then}\ get\_packet?f?l?t?m \rightarrow trans.f.l.t.m \rightarrow \\
\quad\quad\quad ((recack \rightarrow \\
\quad\quad\quad\quad \mathit{if}\ l \;=\; true\ \mathit{then}\ flip \rightarrow conf.ok \rightarrow SenderCtrl \\
\quad\quad\quad\quad\quad\quad\quad \mathit{else}\ advance \rightarrow SendPacket) \\
\quad\quad\quad \Box\ dec \rightarrow SendPacket) \\
\quad\quad \mathit{else}\ abort \rightarrow get\_status?s \rightarrow conf.s \rightarrow SenderCtrl
\end{array}
$$

Fig. 4. The SenderCtrl process

MACHINE *brp_sender*
SEES *Bool_TYPE* , *Bool_TYPE_Ops*
SETS $MESSAGE$; $STATUS = \{\ not\_ok\ ,\ dont\_know\ \}$
CONSTANTS *max_retransmissions*
PROPERTIES $max\_retransmissions \in \mathbb{N}$
VARIABLES
file_left , *send_first* , *send_last* , *toggle* ,
status , *retransmissions*
INVARIANT
$file\_left \in \mathsf{seq}\ (\ MESSAGE\ ) \wedge$
$send\_first \in BOOL \wedge send\_last \in BOOL \wedge$
$toggle \in BOOL \wedge status \in STATUS \wedge$
$retransmissions \in 0\ ..\ max\_retransmissions$
INITIALISATION
$file\_left :\in \mathsf{seq}\ (\ MESSAGE\ )\ \|$
$send\_first :\in BOOL\ \|\ send\_last :\in BOOL\ \|$
$toggle :\in BOOL\ \|\ status :\in STATUS\ \|$
$retransmissions :\in 0\ ..\ max\_retransmissions$
END

Fig. 5. The sender's state definition

packet is sent, and *dont_know* when the protocol aborts during the transfer of the final packet. The constant *max_retransmissions* defines the bound on the number of retransmissions that the sender is prepared to attempt.

The **INITIALISATION** clause assigns each variable non-deterministically[5]. The reason for this apparent freedom is because, in addition to their requirements listed above, the machine operations are defined so that they configure other pieces of state when they are called. Therefore, we do not rely on the initialisation of the machine to configure the state. By using operations to set up the state, we can call these operations to reconfigure the state if necessary (e.g. after the protocol has aborted). The implications of this emerge in the consistency checking performed in Sect. 5. The operations of the sender's B machine are shown in Fig. 6. The **initiate** operation does more than just accept files in readiness for their subsequent transfer: it sets the *send_first* and *send_last* flags, and it also assigns *retransmissions* to the maximum value. Similarly, **get_packet** is responsible for setting the status variable, and **advance** negates the toggle and also assigns *retransmissions* to the maximum value. It is worth noting that the only piece of state that does not get configured at all is the toggle. This implies that the performance of the protocol does not rely on the specific value of the toggle, it merely relies on the toggle's relative value at specific points in its execution.

4.2 The receiver side

In contrast to the *SenderCtrl* process, we can identify three distinct patterns of behaviour in the receiver's controller process: 1) when the receiver is waiting for an initial packet, 2) when the receiver is waiting for subsequent packets (of the same file), and 3) when it is waiting for the first packet of a new file transfer but is also prepared to accept retransmissions of the last packet of the current file. The only piece of state that the receiver depends on is the toggle; this is required to distinguish new packets from

[5] Somewhat confusingly, the operator '||' is used to combine multiple substitutions in B machines.

```
OPERATIONS
  initiate ( ff ) ≙
    PRE ff ∈ seq ( MESSAGE ) THEN
      file_left := ff  ||  send_first := TRUE  ||
      retransmissions := max_retransmissions  ||
      IF size ( ff ) = 1
        THEN send_last := TRUE
        ELSE send_last := FALSE
      END
    END ;

  ff , ll , tt , mm ⟵ get_packet ≙
    PRE size ( file_left ) > 0 THEN
      ff := send_first  ||  ll := send_last  ||
      tt := toggle  ||  mm := first ( file_left )  ||
      IF send_last = TRUE
        THEN status := dont_know
        ELSE status := not_ok
      END
    END ;

  advance ≙
    PRE size ( file_left ) > 0 THEN
      file_left := tail ( file_left )  ||
      send_first := FALSE  ||
      toggle ⟵ NEG_BOOL ( toggle )  ||
      retransmissions := max_retransmissions  ||
      IF size ( tail ( file_left )) = 1
        THEN send_last := TRUE
        ELSE send_last := FALSE
      END
    END ;

  flip ≙
    BEGIN toggle ⟵ NEG_BOOL ( toggle ) END ;

  dec ≙
    PRE retransmissions > 0 THEN
      retransmissions := retransmissions − 1
    END ;

  ss ⟵ get_status ≙ ss := status ;

  rr ⟵ retrans ≙ rr := retransmissions
END
```

Fig. 6. The sender's operations

retransmitted packets. The definition of *ReceiverCtrl* is shown in Fig. 7.

The receiver's B machine is shown in Fig. 8. The operation **initialise** sets the state variable *last_toggle* to the toggle value contained in the first packet that the process *ReceiverCtrl* gets on the *rec* channel. It also provides the first label for the output on the *ind* channel. The operation **get_ind** provides subsequent labels for *ind* and, in addition, updates the value of *last_toggle*. Finally, **examine** determines whether its boolean input matches the value held in *last_toggle*.

4.3 The medium

The process *MEDIUM* defined in Fig. 9 accepts a packet on the *trans* channel and non-deterministically chooses to pass it to the receiver (via the *rec* channel) or lose it by performing a *dec* event. Similarly, it accepts an acknowledgement (by synchronising on the *sendack* channel) and

```
ReceiverCtrl =
  rec?f?l?t?m →
    initialise.f.l.t?i → ind.m.i → sendack →
      if i = last_packet then RECEIVER2
                         else RECEIVER1
  □ abort → ReceiverCtrl

RECEIVER1 =
  rec?f?l?t?m → examine.t?s →
    if s = same then sendack → RECEIVER1
    else get_ind.f.l.t?i → ind.m.i → sendack →
           if i = last_packet then RECEIVER2
                              else RECEIVER1
  □ abort → ind_err → ReceiverCtrl

RECEIVER2 =
  rec?f?l?t?m → examine.t?s →
    if s = same then sendack → RECEIVER2
    else initialise.f.l.t?i → ind.m.i → sendack →
           if i = last_packet then RECEIVER2
                              else RECEIVER1
  □ abort → ReceiverCtrl
```

Fig. 7. The ReceiverCtrl process

```
MACHINE brp_receiver
SEES Bool_TYPE
SETS
  INDICATE =
          { first_packet , last_packet , inter_packet } ;
  COMPARISON = { same , different }
VARIABLES last_toggle
INVARIANT last_toggle ∈ BOOL
INITIALISATION last_toggle :∈ BOOL

OPERATIONS
  ii ⟵ initialise ( ff , ll , tt ) ≙
    PRE ff ∈ BOOL ∧ ll ∈ BOOL ∧ tt ∈ BOOL THEN
      last_toggle := tt  ||
      IF ll = TRUE THEN ii := last_packet
                   ELSE ii := first_packet
      END
    END ;

  ss ⟵ examine ( tt ) ≙
    PRE tt ∈ BOOL THEN
      IF tt = last_toggle THEN ss := same
                          ELSE ss := different END
    END ;

  ii ⟵ get_ind ( ff , ll , tt ) ≙
    PRE ff ∈ BOOL ∧ ll ∈ BOOL ∧ tt ∈ BOOL THEN
      last_toggle := tt  ||
      IF ll = TRUE THEN ii := last_packet
                   ELSE ii := inter_packet
      END
    END
END
```

Fig. 8. The receiver's B machine

$$
\begin{array}{l}
MEDIUM \ = \\
\quad trans?p \rightarrow (rec.p \rightarrow MEDIUM \sqcap dec \rightarrow MEDIUM) \\
\quad \Box \\
\quad sendack \rightarrow (recack \rightarrow MEDIUM \sqcap dec \rightarrow MEDIUM)
\end{array}
$$

Fig. 9. The *MEDIUM* process

non-deterministically chooses to pass it to the sender (by synchronising on the *recack* channel) or lose it by performing a *dec* event. We are able to model this erroneous behaviour naturally because we are using CSP definitions to capture the control flow of the transfer protocol. Note, the protocol assumes that the medium is unable to corrupt packets in any other way (such as by changing the contents of the packet).

5 Formal analysis

By using the CSP processes and B machines defined in Sect. 4, we can combine the various components of the system using parallel composition. That is, we wish to investigate the behaviour of the following system:

$$
\begin{array}{l}
(SenderCtrl \parallel brp\_sender) \\
\parallel MEDIUM \\
\parallel (ReceiverCtrl \parallel brp\_receiver)
\end{array}
$$

Theoretical results enable us to isolate and investigate various sub-components, that are amenable to tool support, to establish properties of the entire system.

The first step in the formal analysis of any combined system is referred to as *consistency checking* (Sect. 5.1). Consistency of a controller/machine pair means that every time a B operation is called, as a consequence of performing a CSP event, its precondition must hold. Otherwise, according to Morgan's semantics (defined in Sect. 2.1), the entire system will diverge. In addition, we must check that all CSP processes are consistent with each other, which means that the parallel composition of the processes cannot reach a deadlock state (i.e. when no event can be performed). Deadlock freedom is a property that can be checked using FDR (Sect. 5.2).

All other properties of interest concern the behaviour of the system at its external interface. For the BRP, these properties refer to the channels *req*, *conf*, *ind* and *ind_err* only. The other channels are abstracted by using CSP hiding operator '\'. However, by hiding such 'internal' events of a process, we introduce the possibility of livelock (i.e. when an infinite succession of internal events can be performed). It is therefore necessary to make sure that hiding does not introduce such undesirable behaviour. Livelock freedom is also a property that can be checked using FDR (Sect. 5.3).

In order to prove properties automatically using FDR, it is usually necessary to augment the controller processes with state information. This is necessary so that the design contracts that exist between a controller and its B machine are made explicit in the process definition. In particular, we add assertions to the events of a process so that the kinds of messages passing between a machine and its controller are made explicit (see Sect. 2.1.1). This prevents an FDR analysis from producing false negatives as counterexamples. Processes augmented with diverging and blocking assertions exhibit additional behaviour whenever an assertion is violated; this behaviour will be detected by FDR. Of course, it is unsafe to introduce assertions that are not fulfilled by the controller/machine pairs. Therefore, each augmentation must be accompanied by a proof of consistency. However, once a property has been established, all such augmentations can be discarded.

5.1 Proof of controller/machine consistency

Consistency in the context of the BRP requires us to show that *SenderCtrl* $\parallel$ *brp_sender* and *ReceiverCtrl* $\parallel$ *brp_receiver* are both divergence-free. In general, given any mutually recursive controller *LOOP* and a machine M, consistency means that $LOOP \parallel M$ is divergence-free or, alternatively, the following property holds:

$$traces(LOOP) \cap divergences(M) = \varnothing$$

This is established if all traces tr of *LOOP* meet the specification $[tr]true$. This specification asserts that any trace tr, when viewed as a sequence of machine operations, is guaranteed to terminate. There will be traces of *LOOP* that are impossible when it is put in parallel with M (since M may not output certain values in the trace), and a notion of *coercion* is used to guarantee that such traces meet the specification miraculously. (See [24] for more details.) Fortunately, such coercions will not mask a divergence if one exists. Section 5.1.1 presents our approach to proving consistency, and Sect. 5.1.2 applies this to the protocol.

5.1.1 A proof technique for consistency

Our method for proving that *LOOP* meets the specification is to construct a predicate called a *control loop invariant* (*CLI*). This is reminiscent of the approach taken in [15] in which a safety property SP is specified in terms of two predicates *Init* and *Etern*:

SP: if a program is started in any state satisfying *Init*, then every state reached during its execution satisfies *Etern*.

This is proved by constructing a predicate I (called an invariant) such that the following three conditions hold:

S1: $Init \Rightarrow I$
S2: every 'atomic' action started in a state satisfying I terminates in a state satisfying I
S3: $I \Rightarrow Etern$

where an 'atomic' action transforms the state (and terminates) without passing through any visible intermediate states.

As is emphasised in [15], control information (the values of 'program counters') is as fundamental as other pieces of state (the values of program variables) because it determines which atomic actions can occur; the invariant I can refer to such control information. For a mutually recursive process $LOOP$, the control information resides in the parameters of its sub-processes. In general, $LOOP$ has the following structure:

$$\begin{aligned} LOOP &= S(x_0) \\ S(x_0) &= R_0 \\ S(x_1) &= R_1 \\ &\vdots \\ S(x_n) &= R_n \end{aligned}$$

where each process body R_i contains recursive calls of the form $S(x_j)$[6] such that $0 \leq j \leq n$. The parameters $x_0, x_1, \ldots, x_n$ provide the necessary control information. As we shall see, the CLI can also refer to such control information.

In order to show that $LOOP$ satisfies the safety property $[tr]true$, a CLI is constructed such that the following conditions are satisfied:

C1: $[\textbf{INITIALISATION}]\,CLI$
C2: $CLI \Rightarrow [BBODY_{R_i}]\,CLI$
C3: $CLI \Rightarrow [tr]true$

where $BBODY_{R_i}$ is the translation of the process body R_i into an equivalent sequence of AMN operations. (See [24] for more details.) The condition C1 corresponds to the hypothesis of SP (with CLI substituted for $Init$) because the starting state is specified by the **INITIALISATION** clause of the B machine. (This substitution also makes S1 trivially true.) C2 corresponds to S2, and C3 corresponds to S3. Notice that, for the purposes of consistency, $BBODY_{R_i}$ is considered to be atomic because we do not need to ensure that the CLI holds for its intermediate states. In fact, Theorem 3.1 of [24] proves that if tr is a trace of $LOOP$ then C3 is consequence of C1 and C2.

We begin by declaring a new variable c_b that is assigned values from the set of parameters $\{x_0, x_1, \ldots, x_n\}$. This variable is called a *control variable* because it is used to keep track of the execution path of the process by highlighting specific *control points*. If c_b is assigned the value x_i then execution of the process body R_i is imminent. The control points of the process are therefore chosen to be the points at which control leaves one process and enters another (although there is no reason why other points within the process bodies cannot be used).

[6] [24] also allows terminating processes.

Without loss of generality, a CLI of $LOOP$ can be defined as follows:

$$(c_b = x_0 \Rightarrow P_0) \wedge (c_b = x_1 \Rightarrow P_1) \wedge \ldots \wedge (c_b = x_n \Rightarrow P_n)$$

where P_i is a predicate on state variables (and possibly c_b) that must hold whenever control of the process enters body R_i. In proving that the AMN translation of R_i maintains the CLI, we observe that since c_b must be assigned the value x_i upon entry to the process body, all but one of the conjuncts in the CLI will be vacuously true at this point. Upon leaving the process body, one of several control points could have been reached (this is due to the branching within a process body). This information is captured by defining two Boolean functions as follows:

- $at(R_i)$ is the assertion $c_b = x_i$ which states that control resides at the entry point of R_i.
- $after(R_i)$ is the disjunction of assertions of the form $c_b = x_j$ if it is possible for the control variable to be assigned the value x_j upon leaving R_i.

The *Locality Rule* from [15] is paraphrased in our setting as follows:

$$\frac{(at(R_i) \wedge CLI) \Rightarrow [BBODY_{R_i}](after(R_i) \wedge CLI)}{CLI \Rightarrow [BBODY_{R_i}]\,CLI}$$

This rule enables us to dispense with the irrelevant conjuncts in the CLI. For example, if $after(R_i)$ is $c_b = x_j \vee c_b = x_k$ then, by using the Locality Rule, the AMN translation of the process body R_i maintains the CLI if we can prove:

$$(c_b = x_i \wedge P_i) \Rightarrow [BBODY_{R_i}]((c_b = x_j \wedge P_j) \vee (c_b = x_k \wedge P_k))$$

5.1.2 Consistency of the BRP

Before we prove consistency of the controller/machine pairs formally, we give the intuition underlying the notion of consistency by using the BRP model defined above. We must show that all of its operations are called within their preconditions. In our example, preconditioned operations in the sender's machine are called from the process *SendPacket* (see Fig. 4). The operation **dec** defined in Fig. 6 requires the remaining number of retransmissions to be positive (i.e. $retransmissions > 0$). However, by virtue of the branching of *SendPacket*, this will always be the case because the query *retrans*?n assigns the current value of *retransmissions* to n, and *dec* occurs within the *then* branch of the conditional $n > 0$. More interestingly, the operations **get_packet** and **advance** both require the state variable *file_left* to be non-empty (i.e. $size(file\_left) > 0$). In other words, these operations require that there is always something to send. There is nothing in the process *SendPacket* to ensure that this will hold when either operation is invoked. Therefore, we infer

that it is necessary to show that whenever control passes to *SendPacket* this piece of state is non-empty.

The two control points of the sender's controller process correspond to the entry points of *SenderCtrl* and *SendPacket*. We therefore assign the control variable to be the value 0 whenever control is at *SenderCtrl*, and assign the value 1 whenever control is at *SendPacket*. The AMN translation (as defined in [24]) converts *SenderCtrl* to give:

ANY $file_b$ **WHERE** $file_b : seq(MESSAGE)$ **THEN**
  IF $\mathsf{size}(file_b) = 0$ **THEN** $c_b := 0$
    ELSE initiate$(file_b)$; $c_b := 1$ **END**

The translation of *SendPacket* gives:

$n_b \leftarrow$ **retrans**;
  IF $n_b > 0$ **THEN**
    $ff_b, ll_b, tt_b, mm_b \leftarrow$ **get_packet**;
      CHOICE
        IF $ll_b = TRUE$ **THEN flip**; $c_b := 0$
          ELSE advance; $c_b := 1$ **END**
      OR
      dec; $c_b := 1$
  ELSE $ss_b \leftarrow$ **get_status**; $c_b := 0$ **END**

From the discussion above, an appropriate choice for the definition of the *CLI* could be $c_b = 1 \Rightarrow \mathsf{size}(file\_left) > 0$ (in which the conjunct for $c_b = 0$ is implicitly given the value *TRUE*). That is, when control enters *SendPacket*, there must be something to send to the receiver. This definition is strong enough to discharge the preconditions of the operations. Unfortunately it is too weak to maintain the *CLI*. In showing that the AMN translation of *SendPacket* maintains the *CLI* using the Locality Rule and the AMN rules defined in [2], we are obliged to prove (after some logical simplification) the following formula:

$$\mathsf{size}(file\_left) > 0 \Rightarrow [\textbf{advance}]\mathsf{size}(file\_left) > 0$$

which, after expanding **advance**, reduces to:

$$\mathsf{size}(file\_left) > 0 \Rightarrow \mathsf{size}(\mathsf{tail}(file\_left)) > 0$$

This is clearly not the case when *file_left* is of length 1. However, control flow to the **advance** operation is also dictated by the last packet flag. We can therefore use the state variable *send_last* to strengthen the definition of the *CLI* as follows:

$$c_b = 1 \Rightarrow (send\_last = TRUE \Rightarrow \mathsf{size}(file\_left) > 0 \wedge send\_last = FALSE \Rightarrow \mathsf{size}(file\_left) > 1)$$

From the AMN translation of *SendPacket*, we can observe that the **advance** operation is called only when *send_last* is *FALSE*. As well as updating the state variable *file_left*, this operation also updates *send_last* according to the size of $\mathsf{tail}(file\_left)$. If this update switches *send_last* to be the value *TRUE* then we have to show that, under the assumption that $\mathsf{size}(file\_left) > 1$, it is the case that $\mathsf{size}(\mathsf{tail}(file\_left)) > 0$; this can, of course, be proven.

Since the *CLI* is stronger than the precondition of **advance**, we can deduce that the sender's controller process calls the operation within a more restrictive set of circumstances than is allowed by the operation itself. However, the precondition was chosen only to permit the safe use of tail and, as such, there is no need to strengthen the precondition to match this restrictive set of circumstances.

5.2 Deadlock freedom

The constraints on the languages of controllers and machines, given in Sects. 2.1.1 and 2.1.2, preclude the occurrence of deadlocks in two respects: non-blocking machines cannot refuse operation calls from the controllers, and the sequential nature of the controller language means that individual controllers cannot deadlock. Therefore, the only potential source of deadlocks is the synchronisation between the controllers (i.e. inconsistencies between the controllers themselves).

The associative and commutative properties of the parallel operator allow us to rearrange the formal model of the BRP by grouping the controllers and the machines as follows:

$$(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \parallel (brp\_sender \parallel brp\_receiver)$$

Therefore, any deadlock of the system will occur within the CSP part of the combination (see Theorem 2 of [20]). This analysis, which is amenable to FDR model checking, indeed confirms that the parallel combination $(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl)$ is deadlock-free. If this check had failed then FDR would provide a trace that causes the refusal of all events in Σ. Such counterexamples can be used to determine flaws in the model.

It is also important, however, to understand what this positive result tells us about the formal model of the BRP. All synchronisations except the *abort* event involve the medium. Once a (non-empty) file has been accepted by the sender, an attempt is made to transmit the first packet on the *trans* channel. At this stage, the medium (shown in Fig. 9 on page 8) is willing to accept the packet, and both processes can proceed: the medium non-deterministically chooses to pass the packet to the receiver or lose the message, and the sender waits for an acknowledgement. However, if the message is lost then the medium is only prepared to perform the *dec* event. Therefore, the sender must not refuse it. Intuitively, this means the sender must not wait for an acknowledgement indefinitely. The model reflects this by having an external choice between the events *recack* and *dec* in the process *SendPacket* shown in Fig. 4.

If a packet is available on the medium, the receiver is always prepared to accept it on the *rec* channel. Once this event occurs, the medium and the receiver proceed by synchronising on the *sendack* channel. The receiver then waits for the next packet to arrive on the *rec* channel. The

only way that the event *rec* can be refused is when the medium decides to *dec*, and since *dec* cannot be refused by the sender, no deadlocks are possible on the receiver's side.

The remaining source of potential deadlocks is the *abort* event. This is unusual because the receiver must be prepared to abort a file transfer even if the first packet of the transfer has not been received (i.e. even if the receiver is unaware that a file transfer has been attempted). This corresponds to the *abort* events in the processes *ReceiverCtrl* and *RECEIVER*2 shown in Fig. 7. Intuitively, the receiver must always be prepared to abort if it is prepared to receive packets.

5.3 Livelock freedom

The proofs of consistency (Sect. 5.1) show that controller/machine pairs do not exhibit divergent behaviour as a consequence of calling an operation outside its precondition. However, as we stated above, by hiding the internal events it is possible to introduce divergent behaviour in the form of an infinite execution of such events. We distinguish between these two forms of divergence by referring to the latter case as a livelock. Intuitively, a livelock should not occur in the BRP because it has bounded behaviour. Therefore, we anticipate the proof of livelock freedom to be dependent on the finite bounds specified in the protocol.

Theorem 3 from [20] allows us to infer livelock freedom of a combined system, whose internal events are hidden, by demonstrating livelock freedom of the controllers.

Theorem 3. *If $P \parallel Q$ is divergence-free, and $C \subseteq \alpha(P)$, and $P \setminus C$ is divergence-free, then $(P \parallel Q) \setminus C$ is divergence-free.*

where $\alpha(P)$ denotes the set of events in the process P.

If we apply this theorem by substituting the parallel combination (*SenderCtrl* $\parallel$ *MEDIUM* $\parallel$ *ReceiverCtrl*) for P and (*brp_sender* $\parallel$ *brp_receiver*) for Q, then by instantiating C with the set of internal events of the BRP model, we can show that (*SenderCtrl* $\parallel$ *MEDIUM* $\parallel$ *ReceiverCtrl*) $\setminus C$ is livelock-free, which is amenable to FDR checking. For the purposes of this analysis (and all subsequent analyses), the set of internal events of the BRP model is defined as follows:

$$C = \Sigma - \{req, conf, ind, ind\_err\}$$

Unfortunately, this livelock freedom analysis fails because the bounded behaviour of the protocol is modelled within the B components (i.e. the finite number of retransmissions that the sender is prepared to make, and the finite length of each file to be transmitted by the sender). Without this information, there is nothing to prevent, say, *SendPacket* and *MEDIUM* synchronising on *trans* and *dec* ad-infinitum because the event *retrans*?*n* can always accept positive values in the absence of *brp_sender*. (By considering CSP processes in isolation, machine channels such as retrans become normal CSP channels which can potentially pass any value.)

We therefore augment the controllers with enough state information in order to show that an infinite succession of internal events is not possible. However, this augmentation must be done so that the behaviour of the controller is preserved in the context of its B machine. We appeal to Theorem 11 of [20] to prescribe how this can be achieved.

State is added to a CSP process by the introduction of parameters that specify how the state changes during its execution. In order to preserve the behaviour of a process, the introduction (or removal) of state must be done so that it does not affect the flow of control. There are two ways in which this can occur: the branching within a process or the recursive calls of a process can be affected by changes to the parameters.

Theorem 11 of [20] is primarily concerned with the safe removal of state from a process through the use of a *collapsing function* which identifies states that are deemed to be equivalent because they yield the same process behaviour. However, this theorem also justifies the introduction of state that is required for our livelock freedom analysis. This is demonstrated by augmenting *SenderCtrl* in the BRP model to produce a process called *SenderCtrl′*. The process *SendPacket* has no parameters. Therefore, all recursive calls to this process (even within *SendPacket* itself) produce the same behaviour (i.e. they all 'jump' to the same place). Providing we do not change any branching within the resulting indexed collection of process definitions, any parameters that are added will not affect the overall behaviour because each component process will have identical behaviour.

We begin by parameterising *SendPacket* with the number of file components that remain to be sent and the number of retransmissions that the sender is prepared to attempt. The call to *SendPacket* is initiated in *SenderCtrl′* with the length of the file that has been accepted on the *req* channel and the maximum number of retransmissions:

$$\begin{aligned}
&SenderCtrl' = req?file \rightarrow \\
&\quad if\ file = <>\ then\ conf.ok \rightarrow SenderCtrl' \\
&\quad else\ initiate{:}file \rightarrow SendPacket(\#file,\ retransmissions)
\end{aligned}$$

By introducing these parameters, we can add diverging assertions to the events retrans and *get_packet* that refer to the current 'state' of *SendPacket* as is shown in the following definition:

$$\begin{aligned}
&SendPacket(x, r) = retrans?n\{n = r\} \rightarrow \\
&\quad if\ n > 0\ then \\
&\qquad get_packet?f?l?t?m\{l \Leftrightarrow x = 1\} \rightarrow trans.f.l.t.m \rightarrow \\
&\qquad\quad ((recack \rightarrow \\
&\qquad\qquad if\ l = true\ then\ flip \rightarrow conf.ok \rightarrow SenderCtrl' \\
&\qquad\qquad else\ advance \rightarrow \\
&\qquad\qquad\qquad SendPacket(x - 1;\ retransmissions))
\end{aligned}$$

$\Box\ dec \rightarrow SendPacket(x, r-1))$
$else\ abort \rightarrow get\_status?s \rightarrow conf.s \rightarrow SenderCtrl'$

Diverging assertions are used at this stage of the analysis because we must perform a consistency check to ensure that the assertions are satisfied in the context of *brp_sender*. Processes augmented with blocking assertions are inappropriate for consistency checking because unsatisfiable assertions do not exhibit divergent behaviour. However, the process is not yet suitable for a livelock freedom analysis because it exhibits divergent behaviour trivially by virtue of the diverging assertions that have been introduced. As these assertions are always true in the context of the B machine (due to consistency), Corollary 1 of [20] allows us to convert them to equivalent blocking assertions, thereby eliminating this source of divergence. Thus, *SendPacket* is defined as follows:

$SendPacket(x, r) = retrans?n\langle n = r\rangle \rightarrow$
$\quad if\ n > 0\ then$
$\quad\quad get\_packet?f?l?t?m\langle l \Leftrightarrow x = 1\rangle \rightarrow trans.f.l.t.m \rightarrow$
$\quad\quad\quad ((recack \rightarrow$
$\quad\quad\quad\quad if\ l = true\ then\ flip \rightarrow conf.ok \rightarrow SenderCtrl'$
$\quad\quad\quad\quad else\ advance \rightarrow$
$\quad\quad\quad\quad\quad\quad SendPacket(x-1, retransmissions))$
$\quad\quad\quad \Box\ dec \rightarrow SendPacket(x, r-1))$
$\quad else\ abort \rightarrow get\ status?s \rightarrow conf.s \rightarrow SenderCtrl'$

Note that one of the parameters strictly decreases with each recursive call to *SendPacket*. Both assertions, which state implicitly that all internal behaviour ceases whenever $r = 0$ or $x = 1$, are fulfilled by *brp_sender* – the proof of consistency confirms this. If we now submit $(SenderCtrl' \parallel MEDIUM \parallel ReceiverCtrl) \setminus C$ to FDR, livelock-freedom is confirmed. Therefore, we can conclude (from Theorem 4.1 of [20]):

$$\begin{pmatrix} SenderCtrl' \parallel MEDIUM \parallel ReceiverCtrl \\ \parallel brp\_sender \parallel brp\_receiver \end{pmatrix} \setminus C$$

is livelock-free. Now we can remove the blocking assertions of *SendPacket* completely (by using Corollary 1 of [20] again). Also, we observe that because the parameters of the process *SendPacket* were introduced so that they did not affect the flow of control, we can safely remove the parameters without changing the process behaviour. This results in the original definition of *SenderCtrl*, and we conclude:

$$\begin{pmatrix} SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl \\ \parallel brp\_sender \parallel brp\_receiver \end{pmatrix} \setminus C$$

is also livelock-free.

6 Composing safety specifications

In this section we investigate more specific patterns of behaviour by using the CSP refinement checking capability of FDR. Such patterns of behaviour are defined as CSP processes – these are called *process-oriented specifications* [21]. Rather than trying to express all desirable patterns of behaviour in a single specification, we define several independent specifications whose 'conjunction' captures an overall notion of desirable behaviour. The 'conjunction' of process-oriented specifications is easily defined in the traces model because it corresponds to parallel composition. For process-oriented specifications R and S, and process P:

$$\text{if } R \sqsubseteq_T P \text{ and } S \sqsubseteq_T P \text{ then } R \parallel S \sqsubseteq_T P$$

where '$\sqsubseteq_T$' denotes the refinement relation in the traces model.We shall construct process-oriented specifications to investigate the safety properties of the BRP. Note, in LOTOS a conjunction of different patterns of behaviour is known as the constraint-oriented specification style, as described in [26].

The events used in process-oriented specifications correspond to the external events of the process under investigation. In Sects. 5.2 and 5.3, the aim is to verify properties of the combined CSP ∥ B system by analysing the CSP components in isolation. Since all external communication is done via the controllers, we once again isolate the controllers in order to prove that they exhibit certain desirable patterns of behaviour (i.e. we use FDR to show that their parallel composition refines the corresponding process-oriented specification). Examining the controllers in isolation is justified by the corollary of Theorem 5 in [20], and is stated as follows:

Corollary. *If a process P has no blocking assertions on any channels of process Q, then $P \setminus \alpha(Q) \sqsubseteq_{SF} (P \parallel Q) \setminus \alpha(Q)$,*

where '$\sqsubseteq_{SF}$' denotes the refinement relation in the stable failures model. For a process-oriented specification *SPEC*, if we can show that $SPEC \sqsubseteq_{SF} P \setminus \alpha(Q)$ by using FDR, then the corollary and the transitivity of the refinement relation prove that $SPEC \sqsubseteq_{SF} (P \parallel Q) \setminus \alpha(Q)$. Note that this result is also valid in the traces model. If we instantiate P with the parallel combination $(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl)$ and instantiate Q with $(brp\_sender \parallel brp\_receiver)$ then the corollary applies only if the internal events of the BRP are machine channels. However, we also want to hide the communication channels linking the controller processes. We therefore instantiate P with:

$$(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \setminus \{trans, rec, sendack, recack, abort\}$$

Since $(P \setminus A) \setminus B =_{SF} P \setminus (A \cup B)$, the corollary is applicable to the BRP once more.

When using CSP process expressions as specifications, it is important to ensure that no acceptable traces are excluded. For example, if the requirement is that the events a and b alternate (beginning with a), then

we might use the process-oriented specification $S = a \rightarrow b \rightarrow S$. If no constraint is required on the other events, then the acceptability of an occurrence of any such event must be stated explicitly. This is achieved by interleaving the constrained behaviour (i.e. S, in this example) with the process $RUN_{\Sigma - \{a,b\}}$, whose traces consist of all possible sequences of events not in $\{a, b\}$.

In the analysis of livelock freedom, we added assertions to certain channels in order to constrain the values that can be passed on those channels. We began by adding diverging assertions and then, after consistency checking, converting them into the corresponding blocking assertions. In the analysis of safety properties using the corollary above, we also need to add assertions to channels. However, we only need to use diverging assertions because divergences are not recorded in the traces model (or the stable failures model). Therefore, the consequence of using such assertions is to remove the behaviours that violate the assertions from the refinement analysis. A proof of consistency demonstrates that such behaviours are impossible in the context of the B machines.

Section 6.1 demonstrates the analysis of one safety property of the BRP. The appendix contains the analyses of three other safety properties. Each analysis follows a similar procedure in order to verify its respective property. Section 6.2 combines these results to form an overall (verified) specification of the protocol. The advantage of composing specifications is to reduce the amount of state information that needs to be lifted in order to verify each property. This makes it easier to identify the source of any unexpected behaviour because it is not obscured by redundant state.

6.1 The buffer properties of the BRP

As with all communications protocols, the aim of the BRP is to overcome the unreliability of the communications medium in order to transmit data from a source to a destination without any corruption. It is natural, therefore, to require that the overall system (comprised of source, medium and destination) exhibits buffer-like properties. These properties, taken from [18, p 114], are stated as follows:

(i) A buffer correctly copies all its inputs to its output channel without loss or reordering.
(ii) Whenever it is empty (i.e. it has output everything that has been input), a buffer must accept any input.
(iii) Whenever it is non-empty (i.e. it has input more than it has output), a buffer cannot refuse to output.

The BRP is different in several respects. First, the input to the BRP consists of a single file, and the output consists of a sequence of file components. Consequently, property (i) must be rephrased to accommodate this alternative notion of copying. Since (ii) and (iii) are liveness properties, we shall not investigate them here. However,

$$Prefix = req?file \rightarrow SendSomeOf(file);\ conf?x \rightarrow Prefix$$

$$SendSomeOf(\langle\rangle) = SKIP$$
$$SendSomeOf(\langle x \rangle^\frown y) = (ind.x?l \rightarrow SendSomeOf(y)) \\ \qquad\qquad \Box\ SKIP$$
$$Error = ind\_err \rightarrow Error$$

$$PrefixSpec = Prefix \ ||| \ Error$$

Fig. 10. A specification of the buffer property of the BRP

the bounded nature of the BRP means that (iii) cannot be achieved because once an abort occurs, the remainder of the input file (at the sender side) is denied the opportunity to be output (at the receiver side).

The process *Prefix* defined in Fig. 10 captures the notion of a partial file transfer by accepting files on its req channel and then, via the process *SendSomeOf*, outputting some arbitrary prefix of the components that make up the file before performing a confirmation event. Notice that, for this property, we are not interested in the labels that accompany the file components (on the *ind* channel), and we do not consider the actual signals that are sent on the *conf* channel; these are investigated in the specifications of the appendix. The occurrences of the *ind_err* event are also of no importance in this specification. Hence, the specification process *PrefixSpec* in Fig. 10 consists of the process *Error* interleaved with *Prefix*.

If we now check whether the parallel combination $(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \setminus C$ is a trace refinement of *PrefixSpec* using FDR, we find that the specification is not met. We need to augment the process with enough state information in order to constrain the data values that are passed on the *ind* channel (i.e. as they would be in the context of the B machines). However, since the process *ReceiverCtrl* merely outputs the data values it receives on the *rec* channel (on the proviso that the toggle is as expected), it is the *SenderCtrl* process that should be constrained because it is responsible for sending the packets via the medium to the receiver.

The buffer property (i) above states that buffer must copy its inputs without loss or reordering. We therefore expect *SenderCtrl* to be augmented with the file to be transmitted. However, the alternating bit plays an important role in ensuring that packets transmitted by the sender are correctly handled by the receiver; the toggle is another piece of state that must be lifted into the *SenderCtrl* process. In fact, both pieces of state are used primarily in the definition of *SendPacket*. In order to constrain the packets sent on the *trans* channel, we add a diverging assertion to the *get_packet* event as follows:

$$SendPacket(t, \langle x \rangle^\frown y) = retrans?n \rightarrow$$
$$\quad if\ n > 0\ then$$
$$\qquad get\_packet?f'?l'?t'?m' \begin{Bmatrix} t' = t \wedge m' = x \\ \wedge (l' \Leftrightarrow \#y = 0) \end{Bmatrix} \rightarrow$$
$$\qquad\quad trans.f'.l'.t'.m' \rightarrow$$

$$
\begin{array}{l}
((recack \rightarrow \\
\quad if\, l' = true\, then\, flip \rightarrow conf.ok \rightarrow SenderCtrl \\
\quad else\, advance \rightarrow SendPacket(\neg t, y)) \\
\quad \Box\, dec \rightarrow SendPacket(t, \langle x \rangle \frown y)) \\
else\, abort \rightarrow get\_status?s \rightarrow conf.s \rightarrow SenderCtrl
\end{array}
$$

The assertion states that the value of the toggle received (t') must be identical to the toggle parameter t, the file component received (m') must be identical to the head the file parameter x, and the last packet flag (l') is true if, and only if, the file parameter consists of a single element (i.e. the tail of the file parameter (y) is the empty list).

However, there is no way to assign an initial value to the toggle t when *SendPacket* is called by *SenderCtrl* because it is initialised non-deterministically in the sender's B machine, and its final value must be negated and retained whenever a file transfer has been successfully completed so that the first component of the subsequent file transfer can be distinguished from the last component of the current file transfer. One solution to the first of these problems is to make an initial call to *get_packet* from *SenderCtrl* in order to get the value of the toggle. This transformation gives us the following process called *SenderCtrl′*:

$$
\begin{array}{l}
SenderCtrl' = req?file \rightarrow \\
\quad if\, file = <> \, then\, conf.ok \rightarrow SenderCtrl' \\
\quad else\, initiate.file \rightarrow \\
\qquad\qquad get\_packet?f?l?t?m \rightarrow SendPacket(t, file)
\end{array}
$$

We can justify this informally by observing that, although *get_packet* changes some state, it does so by assigning constant values to state variables. Hence, calling the operation twice in succession is equivalent to calling it once; we refer to such operations as *idempotent* operations. As long as the maximum number of retransmissions is greater than zero, the operation *get_packet* will be called twice in succession where it had previously been called once. Since all machine events are hidden, the meaning of the process does not change. This novel solution will be discussed in more detail in Sect. 7.

The solution to the second problem is to define an additional process called *SenderCtrl*2. It is almost identical to the original *SenderCtrl* process except it is now parameterised by the toggle t. This process is called whenever a successful file transfer has occurred so that the toggle value is retained for the next file transfer. The complete definition of the sender's augmented controller process is shown in Fig. 11. Notice that this new process definition now resembles the receiver's controller process in that it comprises of three parts. The additional process *SenderCtrl*2 corresponds to *RECEIVER*2 which is needed to distinguish the first component of a new file transfer from the last component of the current file transfer. Of course, such augmentations must respect the definition of the sender's B machine; this is confirmed by a proof of consistency.

The processes *RECEIVER*1 and *RECEIVER*2 of the receiver's controller are also augmented by parameterising them with the a toggle value. This is initialised with the toggle value of the first packet obtained by the receiver. The parameter therefore corresponds to the toggle value of the last packet accepted by the receiver. We name the augmented receiver process *ReceiverCtrl′*.

$$
\begin{array}{l}
SenderCtrl' \;=\; req?file \rightarrow \\
\quad if\;\; file \;=\; <> \;\; then\;\; conf.ok \rightarrow SenderCtrl' \\
\quad else\;\; initiate.file \rightarrow \\
\qquad\qquad get\_packet?f?l?t?m \rightarrow SendPacket(t, file) \\
\\
SendPacket(t, \langle x \rangle \frown y) \;=\; retrans?n \rightarrow \\
\quad if\;\; n \;>\; 0\;\; then \\
\qquad get\_packet?f'?l'?t'?m' \left\{ \begin{array}{l} t' = t \wedge m' = x \\ \wedge (l' \Leftrightarrow \#y = 0) \end{array} \right\} \rightarrow \\
\qquad\quad trans.f'.l'.t'.m' \rightarrow \\
\qquad\quad ((recack \rightarrow \\
\qquad\qquad if\;\; l' \;=\; true\;\; then\;\; flip \rightarrow conf.ok \rightarrow SenderCtrl2(\neg t) \\
\qquad\qquad else\;\; advance \rightarrow SendPacket(\neg t, y)) \\
\qquad\quad \Box\;\; dec \rightarrow SendPacket(t, \langle x \rangle \frown y)) \\
\quad else\;\; abort \rightarrow get\_status?s \rightarrow conf.s \rightarrow SenderCtrl' \\
\\
SenderCtrl2(t) \;=\; req?file \rightarrow \\
\quad if\;\; file \;=\; <> \;\; then\;\; conf.ok \rightarrow SenderCtrl2(t) \\
\quad else\;\; initiate.file \rightarrow SendPacket(t, file)
\end{array}
$$

Fig. 11. The augmented sender controller for *PrefixSpec*

We use FDR to confirm that these augmented processes (when put in parallel with the medium) do indeed combine to give a refinement of *PrefixSpec*:

$$
\begin{array}{l}
PrefixSpec \sqsubseteq_T \\
\qquad (SenderCtrl' \parallel MEDIUM \parallel ReceiverCtrl') \setminus C
\end{array}
$$

6.2 Combining the results

The four safety properties are refined by (different) augmented versions of the parallel process (*SenderCtrl* ∥ *MEDIUM* ∥ *ReceiverCtrl*) \ C. Once these properties have been verified, we can dispense with their respective augmentations by appealing to Corollary 1 and Theorem 11 of [20] (see Sect. 5.3). This converts each augmented *SenderCtrl* and *ReceiverCtrl* process into its original form. As a consequence, we have shown that:

$$
\left(\begin{array}{c} SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl \\ \parallel brp\_sender \parallel brp\_receiver \end{array} \right) \setminus C
$$

refines each of the safety properties. Therefore, this process also refines the parallel composition of these safety properties, which can be viewed as the single process that specifies the overall safety requirements of the BRP. A single refinement would be difficult to prove directly because the controller processes would require considerably more augmentation. This would make it harder to appreciate the different ways in which the state is being used during a run of the protocol.

The parallel composition of a number of specification processes is comparable to Abadi and Lamport's approach to composing specifications [1] in which logical conjunction is used as the composition operator (although their motivation is different).

7 Discussion

CSP ‖ B has been applied in several domains, including information systems [10]. This paper gives a detailed account of the CSP ‖ B approach to the specification and analysis of a communications protocol. Section 7.1 summarises the paper's contribution and the technical issues that have arisen, and Sect. 7.2 positions our integrated approach in relation to other state and event-based approaches.

7.1 New results

In this paper, we specified the expected external behaviour of the BRP via a number of separate process-oriented specifications. This allows the analysis of individual aspects of the system while ignoring others. Once all the specifications have been verified by refinement checking, we can conclude that the parallel composition of the specifications is also proven. This new result complements the existing theory of CSP ‖ B and extends the features of this approach. This has obvious advantages for large systems because it allows us to isolate and investigate subcomponents of the system. We are currently investigating an equivalent notion of composition for liveness specifications. It is not the case that, for process P and failures specifications S_1 and S_2:

if $S_1 \sqsubseteq_{SF} P$ and $S_2 \sqsubseteq_{SF} P$ then $S_1 \parallel S_2 \sqsubseteq_{SF} P$

and there is no other CSP operator that corresponds to this notion of conjunction in the stable failures model.

In addition to compositional verification, several new features have emerged from this CSP ‖ B analysis. The use of *dec* as a machine *and* synchronisation channel (which is a relaxation of previous *CSP* ‖ *B* architectures) was justified informally in Sect. 4.1 because machine channels that call operations as a consequence of synchronisation are more constrained than machine channels that do not require synchronisation. Hence, a proof of consistency that ensures an operation is never called outside its precondition is sufficient to prove that a constrained machine channel never calls an operation outside its precondition. Expressed more formally, if a controller/machine pair $P \parallel M$ is divergence-free and, for a process Q, $\alpha(Q) \cap \alpha(M) \neq \varnothing$ then P also controls Q's calls to M in $(P \parallel M) \parallel Q$ because $\alpha(M) \subseteq \alpha(P)$. Hence, any undesirable behaviour within the system will arise from the CSP component $(P \parallel Q)$ only, which will be discovered by using FDR.

Recall that during the analysis of the safety property in Sect. 6, extra events were added to *SenderCtrl* in order to retrieve the necessary state from the sender's B machine, and we argued that this modification preserved the meaning of the controller/machine pair. However, a certain amount of care is needed when introducing such hidden events. On the CSP side, the introduction of internal events would seem to preserve meaning in all circumstances (since they do not contribute to the observable behaviour of the process). Unfortunately this is not the case because we can introduce non-determinism into an otherwise deterministic process. For example, consider the following process:

$a \rightarrow Stop \,\Box\, b \rightarrow Stop$

If we introduce (and hide) an event c as follows:

$(c \rightarrow a \rightarrow Stop \,\Box\, c \rightarrow b \rightarrow Stop) \setminus \{c\}$

then this is equivalent to

$a \rightarrow Stop \sqcap b \rightarrow Stop$

which, in the stable failures model, is different to the original. On the B side, the operations called as a consequence of these additional internal events should not affect the state; trivially, query operations are examples of such operations.We have introduced the term *idempotent operations* to refer to those operations that change state variables with constant values. Thus, two consecutive calls of an idempotent operation is equivalent to one call of the operation. Of course, consistency must be preserved by the introduction of events.

7.2 Other approaches

Some other notable approaches to the integration of state and event based methods include CSP-OZ [22] and Circus [9]. However, neither of these approaches can utilise existing tool support. For a comparison of these methods with our approach, see [24] and [7] respectively. In this section, we compare CSP ‖ B with Event B, LOTOS, and I/O automata since they have all been used to analyse the BRP.

Analysing protocols is a popular pastime in the event based community because the data passed between protocol participants are usually simple and the potential interactions between the participants can be very subtle. The BRP is unusual because it is a protocol that manipulates data in a non-trivial way, yet it has an intricate flow of control that can only be modelled easily by a process algebra. The analysis has not only given us fresh insights to the protocol (for example, the analysis in Appendix A shows that the first packet flag is redundant), it also gives us insights into the CSP ‖ B approach in general. The formal development consists of three stages: 1) the definition of the specification, 2) the definition of abstract properties, and 3) the verification of the specifi-

cation against these properties. There is no reason why, in practice, the first two stages could not be carried out in either order. However, since we begin with a specific protocol in mind (rather than a set of requirements), it is more natural to proceed in the way prescribed in this paper. The method followed in [13] is similar to ours since it also comprises of separate stages: the definition of the individual I/O automata and their composition, and the identification of invariants of the system.

Our approach involves two well-established formal methods that are at either end of the state/event spectrum. This is because they are based on distinct theoretical ideals: CSP's strength lies in the use of events to model complex behaviour in processes and the interactions that can occur between them. Its weakness lies in the lack of facilities for manipulating state. B, on the other hand, is dedicated to state-based systems with little provision for the control aspects. Even though the state and event-based worlds are in some sense equivalent (see, for example, [1]), the formal models of real world distributed systems rarely fall naturally into either of these two camps, and any attempt to model such systems using formal methods with such strong ideals will usually lead to compromise. However, these ideals have yielded great insights into the kinds of problems that arise in the construction of complex systems, and therefore it would be unwise to abandon them.

It is the aim of CSP ‖ B to retain the ideals on which its component languages are based whilst extending the expressivity of both. Even though distributed systems will typically have rich data combined with a complex flow of control, it is often the case that such features can be modelled separately by defining B machines and CSP processes and then recombined through parallel composition. The only overhead in this approach is the consistency proofs that are needed to ensure that they combine properly.

Nonetheless, attempts have been made to develop formal methods that lie in between the two extremes of the state/event spectrum. Event B uses guarded operations (called events) to implement a notion of flow of control. Here, the enabling of a guard implies the availability of an operation. Thus, it is possible to analyse the sequential execution of operations through the appropriate enabling of guards. Since the B method aims, through refinement, to implement executable code, the unconstrained use of guarded operations is a definite compromise because guards cannot be eliminated. As a consequence, such operations cannot be implemented in general. Also, in Event B (as well as the I/O automata approach of [13]), it is difficult to see the flow of control without analysing the details of all the guards carefully. This has led to the building of compilers such as csp2B [8] to alleviate this problem. In our approach we feel that there is a natural separation of the events under the environment's control and internal events that trigger B operations to cause state updates.

LOTOS is a language whose origins lie in the event based process algebras CCS and CSP. It consists of a set of operators for defining processes and, in addition, introduces a notation for declaring abstract data types. It has a sound theoretical basis but purists would argue that it compromises the ideals on which these separate formalisms are based. The data part of this language is still relatively weak when compared to B as there is no notion of data refinement and, like CSP, state is rather cumbersome because it is passed as a parameter to a process.

In [4], the protocol is constructed following a series of refinements using Event-B. The advantage of this approach is that, in one atomic step, the overall property on the global state can be captured very abstractly using simple predicates. Such a simple abstraction is not possible with our approach because we have to include behavioural information from the outset. Nonetheless, we believe that during the refinement stages it becomes increasingly difficult to clearly visualise the control flow within the protocol which we feel is important for this kind of application. Also, each refinement is specified on a single level, and it is therefore difficult to separate the resources (state variables) among the participants of the protocol. For this application, such a separation is vital in order to reflect the actual resources available in the real world.

The LOTOS analysis of [16] is very similar to ours. The specification of the expected external behaviour and the specification of the protocol itself are both defined as processes. The architecture of the protocol in [16] is also very similar to our CSP architecture (although we define a single process for the medium). Tool support allows the automatic translation of a LOTOS script into a labelled transition system. Then bisimulation is used to show that the protocol specification exhibits its expected external behaviour. This approach could also benefit from compositional verification techniques in order to make the model checking more tractable.

Acknowledgements. Thanks go to Steve Schneider for many discussions on this work. The reviewers' comments are also appreciated. The authors are also grateful to the UK EPSRC for funding under grant GR96859/01.

References

1. Abadi M, Lamport L (1993) Composing Specifications. ACM Transactions on Programming Languages and Systems 15(1):73–132, January 1993
2. Abrial JR (1996) The B Book: Assigning Programs to Meaning. CUP
3. Abrial JR (1996) Extending B without changing it (for developing distributed systems). In: Habrias H (ed) 1st Conference on the B Method, Nantes, November 1996, pp 169–190
4. Abrial JR, Mussat L (1997) Specification and Design of a Transmission Protocol by Successive Refinements using B. Mathematical Models in Program Development 158:129–200. Springer, Nato ASI Series F: Computer and Systems Sciences
5. Behm P, Desforges P, Maynadier JM (1998) METEOR: An Industrial Success in Formal Development. B'98, Montpellier, April 1998, LNCS, vol 1393. Springer

6. Bolognesi T, Brinksma E (1998) Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1):25–29, January 1998
7. Bramble M (2004) Investigating the consistency of combined specifications. MPhil thesis, Royal Holloway, University of London
8. Butler MJ (2000) csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing 12:182–196
9. Cavalcanti A, Sampaio A, Woodcock J (2002) Refinement of Actions in Circus. In: REFINE'02, FMEWorkshop, Copenhagen
10. Evans N, Treharne H, Laleau R, Frappier M (2004) How to Verify Dynamic Properties of Information Systems. In: IEEE International Conference on Software Engineering and Formal Methods, China. IEEE Computer Society Press
11. Havelund K, Shankar N (1996) Experiments in Theorem Proving and Model Checking. In: FME'96, Oxford, March 1996, LNCS, vol 1051. Springer
12. Goldberg A (1983) Smalltalk-80: The Interactive Programming Environment. Addison-Wesley Publishers
13. Helmink L, Selling MPA, Vaandrager FW (1994) Proofchecking a data link protocol. Technical Report CS-R9420, Centruum voor Wiskunde en Informatica (CWI), March 1994
14. Hoare CAR (1985) Communicating Sequential Processes. Prentice Hall
15. Lamport L, Schneider FB (1984) The "Hoare Logic" of CSP, and All That. ACM Transactions on Programming Languages and Systems 6(2):281–296, April 1984
16. Mateescu R (1996) Formal Description and Analysis of a Bounded Retransmission Protocol. INRIA Rapport de recherche 2965
17. Morgan CC (1990) Of wp and CSP. In: Feijen WHJ, van Gasteren AJM, Gries D, Misra J (eds) Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer
18. Roscoe AW (1998) The Theory and Practice of Concurrency. Prentice Hall
19. Schneider S, Treharne H (2002) Communicating B Machines. In: ZB2002, Grenoble, January 2002, LNCS, vol 2272, Springer
20. Schneider S, Treharne H (2002) CSP Theorems for Communicating B Machines. Technical Report CSD-TR-02-12, Dept. of Computer Science, Royal Holloway
21. Schneider SA (1999) Concurrent and Real-Time Systems: the CSP Approach. John Wiley
22. Fischer C (1997) CSP-OZ: A combination of Object-Z and CSP. In: Bowman H, Derrick J (eds) Formal Methods for Open Object-Based Distributed Systems (FMOODS '97), vol 2. Chapman & Hall
23. Stepney S, Cooper D, Woodcock J (2000) An Electronic Purse Specification, Refinement and Proof. Oxford University Computing Laboratory, Technical Monograph PRG-126, July 2000
24. Treharne H (2000) Controlling Software Specifications. PhD Thesis, Royal Holloway, University of London
25. Treharne H, Schneider S, Bramble M (2003) Composing Specifications using Communication. In: ZB2003, Grenoble, June 2003, LNCS, vol 2651, Springer
26. Vissers CA, Scollo G, van Sinderen M, Brinksma E (1991) Specification Styles in Distributed Systems Design and Verification. TCS 89:179–206

A The output on the *ind* channel

The analysis of Sect. 6.1 ensures that the file components output by the *ReceiverCtrl* process are delivered in the correct order with respect to the file input to the sender's side. We now verify that the labels accompanying the file components as they are output on the *ind* channel correspond to the relative positions of the components in the original file. Thus, we want to check that the first component of the file is labelled *first_packet*, the last component of the file is labelled *last_packet* and any other components are labelled *inter_packet*. Since not all file components are guaranteed to get through, the specification must allow the possibility of partial file transfers. This gives us the specification defined in Fig. 12. Notice that we are no longer interested in the contents of the file components, and we consider the labelling relative to the length of the file.

$$
\begin{aligned}
&PrefixLabel = req?file \rightarrow \\
&\quad SendSomeLabels(1, \#file);\ conf?x \rightarrow PrefixLabel \\
&\\
&SendSomeLabels(n, 0) = SKIP \\
&SendSomeLabels(n, 1) = (ind?x.last\_packet \rightarrow SKIP) \\
&\qquad\qquad \Box\ SKIP \\
&SendSomeLabels(n, y) = \\
&\quad if\ n = y\ then\ (ind?x.last\_packet \rightarrow SKIP) \Box SKIP \\
&\quad else\ if\ n = 1\ then \\
&\qquad (ind?x.first\_packet \rightarrow SendSomeLabels(n+1, y)) \\
&\qquad \Box\ SKIP \\
&\qquad else\ (ind?x.inter\_packet \rightarrow SendSomeLabels(n+1, y)) \\
&\qquad\quad \Box\ SKIP \\
&Error = ind\_err \rightarrow Error \\
&\\
&PrefixLabelSpec = PrefixLabel\ |||\ Error
\end{aligned}
$$

Fig. 12. A specification for labelling on the *ind* channel

The first parameter of *SendSomeLabels* corresponds to the position of a particular file component; this ranges from 1 to the length of the file. The second parameter is the length of the file itself. The vacuous case, in which the file length is 0, is defined to be the trivial process *SKIP*. When the length of the file is 1, there can be at most one *ind* event, and this must be labelled *last_packet*. For files of length greater than 1, the labelling is defined with respect to the first parameter.

Once again, we are not interested in the signals passed on the *conf* channel, and we do not care when *ind_err* events occur. Thus, our second specification is defined as *PrefixLabelSpec* in Fig. 12. FDR reports that our original process:

$$(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \setminus C$$

is not a trace refinement of this specification.

The augmentation of *SenderCtrl* for the analysis of this property is similar to that of Sect. 6.1: we add a call to *get_packet* in order to parameterise *SendPacket* with the toggle value t. However, instead of adding the file itself as a parameter of *SendPacket*, we only need to consider its length k. Here, the diverging assertion of $get\_packet?t'?f'?l'?m'$ is $\{t' = t \wedge l' \Leftrightarrow k = 1\}$ which constrains the last packet flag l' to be true only when there remains one file component to be transmitted. Once again, we have to define an additional process *SenderCtrl*2 to retain the toggle value for subsequent file transmissions.

The augmentation of *ReceiverCtrl* is more interesting for this analysis because it is the receiver's B machine

$$
\begin{array}{l}
ReceiverCtrl' = \\
\quad rec?f'?l'?t'?m' \rightarrow \\
\quad\quad initialise.f'.l'.t'?i \left\{ \begin{array}{l} l' \Rightarrow i = last\_packet \\ \wedge\ \neg l' \Rightarrow i = first\_packet \end{array} \right\} \rightarrow \\
\quad\quad\quad ind.m.i \rightarrow sendack \rightarrow \\
\quad\quad\quad\quad if\ i\ =\ last\_packet\ then\ RECEIVER2(t) \\
\quad\quad\quad\quad\quad\quad\quad\quad else\ RECEIVER1(t) \\
\quad \Box\ abort \rightarrow ReceiverCtrl'
\end{array}
$$

$$
\begin{array}{l}
RECEIVER1(t) = \\
\quad rec?f'?l'?t'?m' \rightarrow examine.t'?s\{t' = t \Leftrightarrow s = same\} \rightarrow \\
\quad\quad if\ s\ =\ same\ then\ sendack \rightarrow RECEIVER1(t) \\
\quad\quad else\ get\_ind.f'.l'.t'?i \left\{ \begin{array}{l} l' \Rightarrow i = last\_packet \\ \wedge\ \neg l' \Rightarrow i = inter\_packet \end{array} \right\} \rightarrow \\
\quad\quad\quad\quad ind.m'.i \rightarrow sendack \rightarrow \\
\quad\quad\quad\quad\quad if\ i\ =\ last\_packet\ then\ RECEIVER2(\neg t) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ RECEIVER1(\neg t) \\
\quad \Box\ abort \rightarrow ind\_err \rightarrow ReceiverCtrl'
\end{array}
$$

$$
\begin{array}{l}
RECEIVER2(t) = \\
\quad rec?f'?l'?t'?m' \rightarrow examine.t'?s\{t' = t \Leftrightarrow s = same\} \rightarrow \\
\quad\quad if\ s\ =\ same\ then\ sendack \rightarrow RECEIVER2(t) \\
\quad\quad else\ initialise.f'.l'.t'?i \left\{ \begin{array}{l} l' \Rightarrow i = last\_packet \\ \wedge\ \neg l' \Rightarrow i = first\_packet \end{array} \right\} \rightarrow \\
\quad\quad\quad\quad ind.m'.i \rightarrow sendack \rightarrow \\
\quad\quad\quad\quad\quad if\ i\ =\ last\_packet\ then\ RECEIVER2(\neg t) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ RECEIVER1(\neg t) \\
\quad \Box\ abort \rightarrow ReceiverCtrl'
\end{array}
$$

Fig. 13. The augmented *ReceiverCtrl* for *PrefixLabelSpec*

that is responsible for labelling the ind channel. As before, we add the toggle as a parameter to the processes *RECEIVER*1 and *RECEIVER*2. However, now we need to add diverging assertions to the *initialise* events of *ReceiverCtrl* and *RECEIVER*2, and add a diverging assertion to the *get_ind* event in *RECEIVER*1. This is defined as the process *ReceiverCtrl′* shown in Fig. 13. These assertions state the relationship between the label (l') and the value of the *last_packet* flag that has been received (i): if the value of the flag is *false* then the label must be *first_packet* in the processes *ReceiverCtrl′* and *RECEIVER*2, and it must be *inter_packet* for *RECEIVER*1. (If the flag is true, the label must be *last_packet* in both cases.) As before, a proof of consistency is required to ensure that the receiver's B machine does indeed output these values.

Now we can verify the trace refinement of the specification by using FDR. It is interesting to note that this result tells us that the label on the *ind* channel can be determined precisely by looking at the last packet flag. This demonstrates that the first packet flag is completely redundant in the protocol.

B The relationship between *conf* and *ind*

We now specify the nature of the signals that are output on the sender's *conf* channel by relating them to the labels that accompany the file components on the receiver's

$$
\begin{array}{l}
ConfSignal = req?file \rightarrow \\
\quad if\ \#file = 0\ then\ conf.ok \rightarrow ConfSignal \\
\quad else\ (if\ \#f = 1\ then \\
\quad\quad\quad (conf.dont\_know \rightarrow ConfSignal \\
\quad\quad\quad\ \Box\ SendSomeInds;\ ConfSignal) \\
\quad\quad else \\
\quad\quad\quad (conf.not\_ok \rightarrow ConfSignal \\
\quad\quad\quad\ \Box\ SendSomeInds;\ ConfSignal))
\end{array}
$$

$$
\begin{array}{l}
SendSomeInds = ind?x?l \rightarrow \\
\quad ((if\ l = last\_packet\ then \\
\quad\quad\quad (conf.ok \rightarrow SKIP\ \Box\ conf.dont\_know \rightarrow SKIP) \\
\quad\quad else\ (conf.not\_ok \rightarrow SKIP\ \Box\ conf.dont\_know \rightarrow SKIP)) \\
\quad \Box\ SendSomeInds)
\end{array}
$$

$$Error = ind\_err \rightarrow Error$$

$$ConfSignalSpec = ConfSignal\ |||\ Error$$

Fig. 14. A specification for correct signalling of *conf*

ind channel. The previous analysis establishes that the labelling accurately reflects the relative position of the file components output by the receiver. Therefore, by deriving a similar correspondence between *conf* signals and *ind* labels, we aim to show that the signal event issued by the sender controller on termination of a particular file transfer (either successfully or unsuccessfully) is also accurate.

The length of the input file also impacts on the signals that are possible: if a file is empty nothing is sent over the medium. Hence, *conf.ok* should be the only outcome. If the file contains one packet then either the whole file is sent, or its packet is not sent at all (before an abort occurs). Thus, the only outcomes are *conf.ok* or *conf.dont_know*. If the file contains two packets then, in addition, one packet can be sent (i.e. all packets minus the final packet) before an abort occurs. Thus, there is the additional possibility of *conf.not_ok*. If the file contains more than two events then, in addition, any other non-empty prefix of the file can be sent before an abort occurs.

This specification is defined as the process *ConfSignal* in Fig. 14. The process *SendSomeInds* simply outputs a sequence of *ind* events followed by a single *conf* event. The signal that accompanies this event is determined from the label in the final *ind* event: if the label is *last_packet* then the signal is either *ok* or *dont_know* (which depends on whether the final acknowledgement is received). Otherwise, the signal is either *not_ok* or *dont_know* (which depends on whether the final packet reaches the receiver, and no subsequent acknowledgement is received).

The trace refinement of this property is not met by $(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \setminus C$ without augmenting *SenderCtrl* and *ReceiverCtrl*. However, the reasons for introducing state are more subtle than the previous analyses. Since we are no longer interested in the contents of the file or the relative position of its file

$$
\begin{array}{l}
SenderCtrl' \;=\; req?file \rightarrow \\
\quad if\ file = <>\ then\ conf.ok \rightarrow SenderCtrl' \\
\quad else\ initiate.file \rightarrow \\
\qquad get\_packet?f?l?t?m\{l \Leftrightarrow \#f = 1\} \rightarrow SendPacket(t, l)
\end{array}
$$

$$
\begin{array}{l}
SendPacket(t, l) \;=\; retrans?n \rightarrow \\
\quad if\ n > 0\ then\ get\_packet?f'?l'?t'?m'\{t' = t \wedge l' = l\} \rightarrow \\
\qquad trans.f.l.t.m \rightarrow \\
\qquad ((recack \rightarrow \\
\qquad\quad if\ l' = true\ then\ flip \rightarrow conf.ok \rightarrow SenderCtrl2(\neg t) \\
\qquad\quad else\ advance \rightarrow \\
\qquad\qquad get\_packet?f''?l''?t''?m'' \rightarrow SendPacket(\neg t, l'')) \\
\qquad \Box\ dec \rightarrow SendPacket(t', l')) \\
\quad else\ abort \rightarrow get\_status?s \left\{ \begin{array}{l} l \Rightarrow s = dont\_know \\ \wedge\ \neg l \Rightarrow s = not\_ok \end{array} \right\} \rightarrow \\
\qquad conf.s \rightarrow SenderCtrl'
\end{array}
$$

$$
\begin{array}{l}
SenderCtrl2(t) \;=\; req?file \rightarrow \\
\quad if\ file = <>\ then\ conf.ok \rightarrow SenderCtrl2(t) \\
\quad else\ initiate.file \rightarrow \\
\qquad get\_packet?f'?l'?t'?m'\{t' = t \wedge l' \Leftrightarrow \#f = 1\} \rightarrow \\
\qquad\quad SendPacket(t', l')
\end{array}
$$

Fig. 15. The augmented *SenderCtrl* for *ConfSignalSpec*

components, it is perhaps surprising that the toggle is required as a parameter of both processes. The reason is to ensure that both the sender and the receiver agree on the acknowledgements that are sent and received. For example, without the toggle to constrain the behaviour of the processes it is possible for the sender to receive every acknowledgement (and, hence, progress with the transfer of subsequent file components), yet the receiver can believe that every packet received is a retransmission of the first packet and, consequently, send acknowledgements without delivering the file components. This results in the sender believing that a successful file transfer has occurred, whilst the receiver believes that only the first component actually got through. In this case, the *conf.ok* signal will not be accurate.

A second piece of state is added to the sender's controller. It corresponds to the last packet flag contained in the packet that is currently being transmitted by the sender. This is needed to constrain the status values that can be received by the *get_status* event in the *abort* branch of *SendPacket* because the *conf.not_ok* and *conf.dont_know* signals are determined from the sender's B machine (see Sect. 4.1). Without this parameter, this information is not available in this branch of the process and, hence, it would not be possible to make such a constraint. The augmented definition of *SenderCtrl* is shown in Fig. 15. Note, the *get_packet* event introduced after the *advance* event is another example of the introduction of idempotent events mentioned above (and discussed further in Sect. 7). This is necessary so that the *last_packet* parameter is maintained as the file transfer progresses.

$$
\begin{array}{l}
Error \;=\; conf.not\_ok \rightarrow Error \\
\qquad \Box\ conf.dont\_know \rightarrow Error \\
\qquad \Box\ ind?x?l \rightarrow (ind\_err \rightarrow conf.not\_ok \rightarrow Error \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad conf.not\_ok \rightarrow ind\_err \rightarrow Error) \\
\qquad \Box\ ind?x?l \rightarrow (ind\_err \rightarrow conf.dont\_know \rightarrow Error \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad conf.dont\_know \rightarrow ind\_err \rightarrow Error)
\end{array}
$$

$$Inds = ind?x?l \rightarrow Inds$$

$$Reqs = req?f \rightarrow Reqs$$

$$OKs = conf.ok \rightarrow OKs$$

$$ErrorSpec = Error \;|||\; Inds \;|||\; Reqs \;|||\; OKs$$

Fig. 16. A specification for *ind_err* events

C The Properties of *ind_err*

We would also like to establish a correspondence between the occurrences of *ind_err* on the receiver's side and the occurrences of *conf* signals on the sender's side. For example, it is reasonable to expect that there should be no *ind_err* event whenever a *conf.ok* occurs. Also, there should be at most one *ind_err* event for each unsuccessful file transfer (signalled by the events *conf.not_ok* or *conf.dont_know*). However, the former case is not true. The reason is due to the allowance of 'empty' file transfers (i.e. the transfer of files of length 0 – a situation that is not allowed in [16]). It is possible for the sender and receiver to abort a run of the protocol whereupon the sender performs a *conf.not_ok* event. If the receiver is slow in performing the corresponding *ind_err* event, the sender can engage in arbitrary many 'empty' file transfers, each resulting in a *conf.ok* signal. When the receiver chooses to perform its outstanding *ind_err* event, this requirement is violated. One solution to this problem is to forbid the 'transfer' of empty files (as in [16]). However, we choose to weaken the requirements in order to ascertain what can be said about the properties of *ind_err* in this model of the BRP. The weaker specification is formalised in Fig. 16.

This specification states that the occurrence of an *ind_err* event must be accompanied by a *conf.not_ok* event or a *conf.dont_know* event and, in either situation, these events must have been preceded by at least one *ind* event. It is possible, however, for either *conf.not_ok* or *conf.dont_know* to occur without an associated *ind_err* being issued from the receiver side. This happens in the case when no packets reach the receiver. The result of the FDR analysis shows that the specification is refined by $(SenderCtrl \parallel MEDIUM \parallel ReceiverCtrl) \setminus C$ without any augmentation.