Modelling and Analysis of the Bounded-Retransmission Protocol: Experience with Discrete Time in the LTSA

Dimitra Giannakopoulou

Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK
dg1@doc.ic.ac.uk

Abstract. The Bounded Retransmission Protocol is an industrial protocol designed for the transfer of large data files over unreliable communication lines. The protocol relies on specific assumptions on the timed behaviour of its components. This paper describes our experience with modelling and analysing the Bounded Retransmission Protocol using the LTSA. The LTSA uses labelled transition systems to specify behaviour, and compositional reachability analysis to incrementally generate, minimise, and analyse a system, based on its software architecture. The tool was not originally designed to deal with real-time applications. However, by modelling time as a discrete entity, the LTSA does not need to be extended in order to handle such systems. We discuss how the features of the tool can be exploited to model and analyse behaviours that involve time.

1 Introduction

The Bounded Retransmission Protocol (BRP) is a non-trivial extension of the Alternating-Bit Protocol, designed for the transfer of large data files over unreliable communication lines. BRP is a simplified variant of a telecommunication protocol used in one of Philips' products. The protocol bases its correctness on specific assumptions about the timing aspects of a system. It has attracted the interest of the research community because it falls beyond the size of classical academic case studies, and because it involves challenges in such issues as finding appropriate abstractions and checking assumptions that involve time. As far as the timing assumptions of the protocol are concerned, there have been two main approaches to solving the problem. First, the protocol may be modelled in an "un-timed" way, in which case the timing assumptions on which its correctness is based need to be introduced as constraints in the system behaviour [1-4]. Second, time may be modelled explicitly in the protocol. This is more challenging, but establishes a global view of the way the protocol works, and of the factors that may affect its correctness [5]. We use the timed version of the BRP case study as a way of estimating the capabilities of our current approach and toolset in dealing with non-trivial timed systems.

The main objective in the development of our analysis methods and tools is to make these accessible to practising software engineers. We have therefore aimed at techniques that form an integral part of the software development process, that are intuitive to use, and that are supported by powerful automated tools. Our approach supports analysis directed by the software architecture of a system [6-8]. In particular, it is based on the use of Labelled Transition Systems (LTS) to specify behaviour and Compositional Reachability Analysis (CRA) to check composite system models. The architecture description of a system drives CRA in generating the model of the system based on models of its components. The model thus generated can be checked against the properties required of it. Previous papers have addressed the problem of verifying safety and liveness properties in the context of CRA [9, 10], as well as a special class of properties that we

call progress [11]. We have also proposed a novel way of dealing with fairness in an intuitive and efficient way, by combining a fair choice assumption, and an action priority scheme [11].

Our tools were not originally designed to handle time. However, we found that, without extending them, it is possible to model timed systems in a straightforward fashion. This is achieved by modelling time as a discrete entity, and making processes that need to count time synchronise on a global "tick" action [12]. Our approach to fairness allows us to elegantly model the assumption of maximal progress usually made on timed systems. Moreover, the integrity of a model with respect to time can be checked by using a simple progress property. On the other hand, counting time in terms of clock ticks is error-prone, and may increases the size of a model significantly. Additionally, as will be discussed in the paper, introducing time in the context of CRA raises a number of issues.

This paper provides a detailed description of our approach to the design of distributed systems, reflected through the BRP case study. Section 2 presents the problem that the protocol is designed to solve, and lists informally the requirements against which the protocol is to be checked. Section 3 describes the software architecture of the system, and then discusses the behaviour of the protocol in terms of its architectural components. The timing constraints are also introduced and a first estimate is made on the values of timers that must be used for these constraints to hold, given the software architecture presented. Section 4 reports on our experience with modelling the protocol and its properties, whereas section 5 concentrates on the analysis of the model developed. Analysis is performed in two stages. First, to show that the correctness of the protocol depends on the timing assumptions that we have set, we check the protocol without taking time into account. Second, we show that, given specific constraints on the timer values of the protocol, all properties required of it are satisfied. Finally, section 6 closes the paper with conclusions and plans for future work.

2 Requirements

Figure 1: Structural requirements for a file transfer service

The Bounded Retransmission Protocol (BRP) provides a transfer service for large data files within a limited amount of time. As illustrated in Figure 1, the protocol implements a component that is in charge of transferring files from a sending to a receiving client, S_CLIENT and R_CLIENT, respectively. It receives a file through port *accept*, breaks the file up into small data chunks, and delivers these chunks in order through port *deliver*. The protocol operates over lossy FIFO channels. As a result, chunks may get lost during transmission. Given the time limitations for file transfer, only a bounded number of retransmissions are allowed for each chunk. If the protocol is unsuccessful in transmitting any chunk of a file, transfer is aborted. The receiving client thus receives a (possibly empty) prefix of each file to be transferred. Transfer is considered successful when this prefix equals the entire file. The protocol is required to provide appropriate indications to both clients about the result of the transfer. Port "result" is used to send indications to the sender, whereas the receiver gets indications within the chunks delivered through port "de-

liver". Note that, the protocol may not be able to conclude whether the last chunk of a file was transferred successfully, in which case it reports a "don't know" result to the S_CLIENT. In all other cases, the indications provided by the protocol are conclusive.

The requirements for the protocol can be informally summarised as follows:

- the chunks of a file sent by S_CLIENT are delivered to R_CLIENT in order and at most once each;
- if S_CLIENT gets an indication that transfer of a file was successful, then all chunks of that file have been delivered to R_CLIENT;
- if S_CLIENT gets an indication that transfer of a file was unsuccessful, then at least one chunk has not been delivered to R_CLIENT;
- R_CLIENT gets an indication that a file has been transferred successfully, *if and only if* it has received all chunks of that file;
- if R_CLIENT receives at least one chunk from a specific file, it can eventually conclude about the result of the transfer (i.e., it knows whether the transfer was completed or not);
- the protocol is always eventually ready to receive a new file from S_CLIENT;
- for each file that it submits to the protocol, S_CLIENT eventually receives a result regarding its transmission.

The last two properties involve liveness issues; they force the protocol to do something useful, rather than checking that it does not do something wrong, which is the case with safety properties. In the following, we describe BRP, and prove that it satisfies the above requirements under strict timing constraints.

3 The Bounded Retransmission Protocol (BRP)

This section provides several descriptions of the protocol, at different levels of abstraction. First of all, we discuss the software architecture of the protocol in our Architecture Description Language (ADL) Darwin [13]. Subsequently, we describe the way the protocol works, in terms of the functionality of its components.

3.1 Software Architecture

The BRP is an extended version of the Alternating-Bit Protocol (ABP), designed to deal with two additional requirements: (i) it is allowed a limited number of retransmissions, and (ii) it handles the transfer of large files, which it breaks down into smaller chunks. The ABP and BRP have identical software structures. In fact, only small variations need to be applied to the behaviour of the architectural components of their common structure to obtain one or the other protocol.

As illustrated in the architecture of Figure 2, the protocols run a transmitter process on the S_CLIENT's site, and a receiver process on the R_CLIENT's site. Given the fact that Darwin does not distinguish between components and connectors, we include the channels over which the sender and receiver communicate as components in the architecture. As illustrated in Figure 3, both the transmitter and the receiver are further decomposed into a timer component each, in addition to components TX and RX that provide their basic functionalities, respectively.

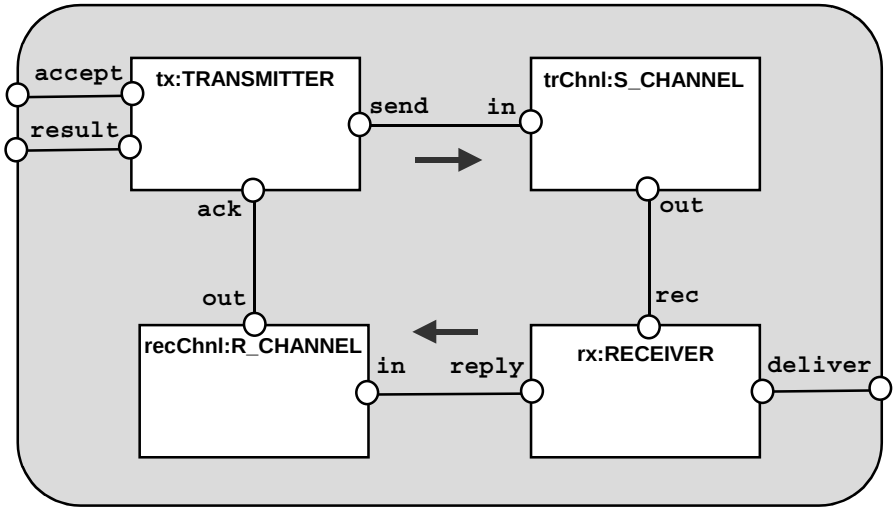

Figure 2: Software Architecture of ABP and BRP protocols

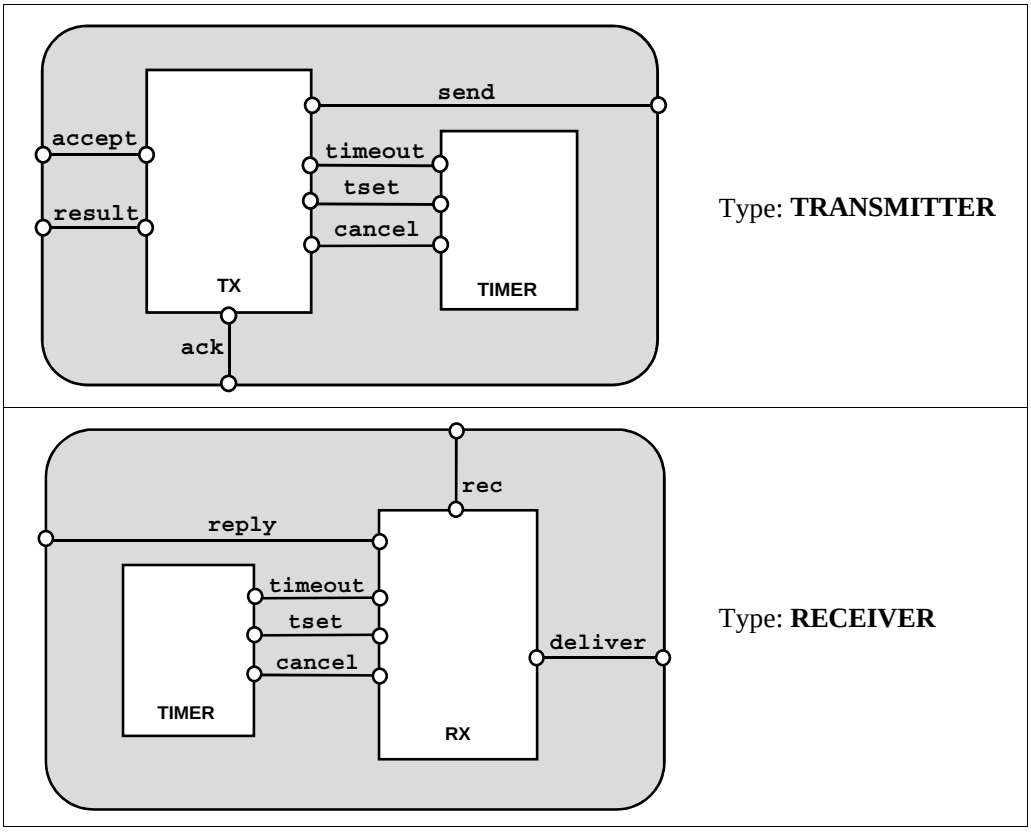

Figure 3: Structure of transmitter and receiver components

3.2 Protocol description

This section describes BRP informally, in terms of the functionalities of the components illustrated in Figure 2 and Figure 3. The indications that TX reports to S_CLIENT through port *result* are the following:

- **ok** – denotes that file transfer has been completed successfully;
- **nok** – denotes that the transfer has been aborted;
- **don't know** – the transfer may have completed successfully, or the last frame may have got lost.

The last indication reflects the fact that the protocol may not be able to give a definite result about the outcome of the transfer (result "don't know"). This situation arises because no realistic implementation can conclude in all cases whether the last chunk was lost or not. The information about a successful delivery (ack) has to be transmitted over an unreliable communication medium. Since a limited number of retransmissions are allowed, there is no way of knowing whether no acknowledgement means that the message was not delivered, or that the ack got lost.

Component RX attaches the following indications to the data that it sends to R_CLIENT:

- **first** – this is the first frame of a file;
- **int** – this is an intermediate frame;
- **ok** – this completes the transfer;
- **nok** – transfer was aborted.

The protocol receives a file through port *accept* of component TX, which breaks the file up into frames. It then starts transmission of chunks (port *send*), where each chunk consists of:

(i) a frame of the file;
(ii) a bit *ab* that implements an alternating-bit scheme, used by the receiver to distinguish new messages from retransmissions;
(iii) a bit *f* that indicates whether this is the first frame of the file;
(iv) a bit *l* that indicates whether this is the last frame of the file.

Note that, if both *f* and *l* are set, then the file consists of a single frame, and therefore the frame sent is both the first and the last one of the file. On sending a chunk to RX through S_CHANNEL, TX sets its TIMER to value T_1, and waits either for an acknowledgement (port *ack*) or for a *timeout* to occur. In the former case, TX knows that the message has been transmitted successfully. If this was the last frame of the file, it reports "ok" to S_CLIENT. If a timeout occurs, it assumes that either the message, or its acknowledgement has been lost, and retransmits the same chunk. This happens unless the maximum number of allowed retransmissions has been reached, in which case TX decides to abort the transfer of the file. It then reports to S_CLIENT "don't know" if the message just sent contained the last frame of the file, and "nok" otherwise.

The receiver remembers whether the previous frame was the last one of a file and the expected value of the alternating bit *ab* for new messages. Note that, on the receiver's side, if the previous frame was the last of a file, then the next new message can only be the *first* frame of the subsequent file. On receipt of a message, RX checks the value of its *ab*. If *ab* has the expected value, then RX delivers it to R_CLIENT (port *deliver*), with the appropriate indication (if the chunk received is the last of the file, the indication is "ok"). If *ab* has a different value than the expected one, then RX assumes that the chunk received is a retransmission, and does not deliver it. In both cases, RX sends an acknowledgement to TX through channel R_CHANNEL (port *reply*), and sets

its timer to some value T_2. It then expects either a new message, or a timeout to occur. A new message is tackled as described above. A timeout indicates that file transfer was aborted at the transmitter's end. Provided that the last chunk of the file has not just been delivered, RX informs R_CLIENT that the transfer was given up ("nok"). In any case, if a timeout occurs, RX must synchronise again with TX on the alternating-bit scheme. It does this based on the *ab* value of the next message that it receives from TX. Note that, if the first frame of a new file is received before the timer expires, the alternating bit scheme is simply continued.

As mentioned, the alternating-bit scheme is used to deal with duplicate messages. We briefly describe how the scheme works in terms of an example. Assume that TX sends a message with *ab* = 1. RX expects the new message from TX to be tagged with this value. If the message is lost on its way, TX retransmits the message with *ab* = 1. If RX receives this second message, it knows that this is a message not yet delivered (since it bears the expected value for *ab*). It delivers the message to R_CLIENT, issues an ack, and changes the expected *ab* value to *0*. If the ack is lost, TX retransmits the message. On receipt of the latter message, which is tagged with *ab*=1, the receiver does not deliver it; it knows that this is a retransmission, because the expected *ab* value for new messages is *0*.

The following property characterises the correct synchronisation of transmitter and receiver on the alternating-bit scheme:

- at the receiver's end, any consecutive messages tagged with the same bit, must bear identical contents, since they are considered to be retransmissions.

This is a general property that applies to any protocol using the alternating bit scheme. For example, we introduced this property in the analysis of the ABP [14].

3.3 Timing constraints for correctness

The protocol works correctly under strict timing requirements that reflect the following two assumptions:

(i) *timeouts do not occur prematurely*. In other words, after TX sends a message, its timer expires iff either the message was lost on S_CHANNEL, or its corresponding acknowledgement was lost on R_CHANNEL. Additionally, when RX is in the process of receiving a file, its timer expires iff transfer has been (or will definitely be) aborted by TX.

(ii) *if transmission of a file is interrupted, RX must not receive any chunk from a new file before realising this fact.* This is necessary for TX and RX to remain synchronised on the alternating bit scheme. We show in our section on analysis how the scheme can be broken if this assumption is not satisfied.

The above assumptions are implemented by selecting appropriate values to which the timers used by the protocol are set. In what follows, we estimate what the minimal values must be for these requirements to hold. In the next section, we prove formally the correctness of the protocol given these values.

Assumption (i)

Let us denote with CD the maximum delay of the channels, with TRIES the maximum number of transmissions allowed for the same message (i.e. TRIES-1 retransmissions allowed), and let us assume that the time taken by RX to process a message is negligible. If no premature timeouts are to occur on the transmitter's side, then the value T_1 to which it sets its timer must be greater

than the maximum time it takes for an acknowledgement (corresponding to the message sent) to be received. Then obviously, $T_1 > 2*$ CD, which makes the minimum value for T_1 : 2* CD + 1.

Additionally, we must guarantee that T_2 expires only after it is certain that the current file transfer is to be aborted. In other words, T_2 must exceed the maximum delay between the moment that it sends an acknowledgement (this is when it resets its timer), and the moment where it is guaranteed that TX is going to abort (if it has not already done so). The worst case for this delay is illustrated on the left of Figure 4, which describes the following scenario. Component TX receives the ack to some chunk of the file CD time units after RX set its timer. Subsequently, TX makes TRIES attempts at sending the next chunk, but all of them are unsuccessful because S_CHANNEL loses the messages (if a message is received, RX will reset its timer). We know that TX attempts a new retransmission T_1 time units after the previous one. If RX has not received the chunk CD time units after the last retransmission, TX is bound to abort the transfer, since there is obviously no way it can receive an ack for this chunk. From the diagram of Figure 4, we can thus conclude that:

$$T_2 > CD + (TRIES-1)* T_1 + CD = 2*CD + (TRIES-1)* T1.$$

The minimum value for T_2 is thus: $2*CD + (TRIES-1)* T_1 + 1$. With T_1 assuming its minimum value calculated above, we end up with the following minimal value for T_2: TRIES * T_1.

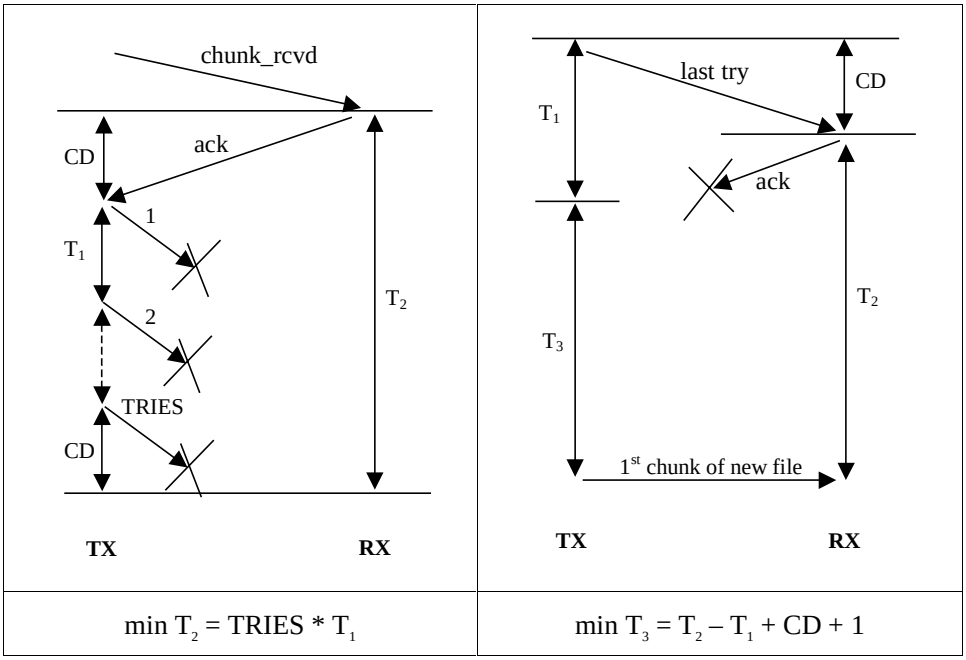

Figure 4: Minimal timing constraints for BRP correctness

Assumption (ii)

This is necessary for TX and RX to remain synchronised on the alternating bit scheme. We show in our section on analysis how the scheme can be broken if this assumption is not satisfied.

To make sure that this assumption holds, we introduce a delay in the behaviour of TX, between the moment it decides to interrupt the transfer of the current file, and the moment of accepting a new file to transmit. After deciding to abort, TX thus sets its timer to some value T_3,

and waits for it to expire before starting a new file transmission. The minimal value for T_3 is determined by the scenario illustrated on the right of Figure 4. According to that, $T_3 > CD + T_2 - T_1$, therefore min $T_3 = T_2 - T_1 + CD + 1$.

4 Modelling

We proceed by modelling the protocol in the language of our analysis tool, FSP [12]. We model the components based on the Software Architecture of Figure 2, and respect the interface names [7, 8, 15]. Modelling helps to understand the subtle details of the protocol. Additionally, the model generated may be used to establish correctness with respect to the requirements set in Section 2. This is performed in an automated way, by using model checking techniques.

One of the things that model checking can demonstrate on our example is that BRP is highly dependent on the timing assumptions stated in the previous section. Note that we do not wish to introduce the timing constraints using "synchronisation tricks" that will control the occurrence of timeouts in one component, by monitoring the occurrence of some events in other components (this is the approach used in the un-timed verifications of the protocol). Our aim is to perform analysis without affecting the modularity and structure established in the protocol's software architecture. For this reason, we introduce timing constraints by selecting appropriate values for timers, as is typically done in such a distributed environment. In addition, it is obvious that for this protocol to operate correctly, some knowledge must be available about maximum channel delays and processing speed of the components. These play a crucial role in establishing values for timers, and needs to be reflected in our model.

We model time as a discrete entity, as currently supported by our methods and tools. In what follows, we discuss the experience we obtained by facing several choices and pitfalls in modelling BRP, a protocol that is far from trivial.

4.1 Constants & Ranges

The following constants and ranges are used in our specification of the protocol. Note that the timer values T_1, T_2 and T_3 are set to the minimal values calculated above.

```
const TRIES = 2                  // total number of transmissions
const FILE_LENGTH = 3            // #chunks in a file
const CD = 2                     // max channel delay

range BIT = 0..1
range CHUNKS = 1..FILE_LENGTH    // chunk ids
range DELAYS = 0..CD             // possible channel delays

set LABELS = {first, int, ok, nok}
set BOOL = {true, false}

const T1 = 2*CD + 1
const T2 = T1 * TRIES
const T3 = T2 - T1 + CD + 1
```

We make several abstractions in order to avoid generating an unnecessarily large state space. First of all, we do not include the contents of frames to the messages. Rather, we characterise

each frame by its sequence in the file, i.e. frame ids fall within range CHUNKS (values from 1 to FILE_LENGTH). Due to the fact that the protocol behaviour does not depend on the contents of the values that are transmitted, it is sufficient to check its correctness for FILE_LENGTH = 3 [16]. However, we can easily change the value of this constant, and we have indeed checked correctness for several values of FILE_LENGTH. Note that, although the id of a chunk gives its exact position within a file, the protocol behaviour does not make use of this information. As mentioned in the description of the protocol, it is the various bits added to data sent that are used to communicate all the necessary information in the protocol behaviour. The chunk sequence numbers are only used in our model to simplify the state space, and in order to make the verification task easier.

We have checked the protocol for various values of the CD and TRIES constants. DELAYS is a range that describes the potential delays that a channel may exhibit. Set LABELS contains the indications that the receiver appends to chunks before delivering them to the R_CLIENT. Finally, timers are set to their minimal values, as described previously.

4.2 Components and time

Our tools currently support the possibility of modelling time as a discrete entity. This choice is motivated by the fact that this approach to dealing with time does not require additions to our tools. Moreover, an un-timed model (where timeouts would typically occur non-deterministically) can be easily extended to incorporate timing aspects either for simulation purposes only [17] or by modifying specific components such as timers to deal with time explicitly. We hope that with extensive experimentation within the current framework, we will be able to decide on what additional modelling and analysis techniques we may wish to introduce in order to facilitate the process of dealing with time. In particular, we wish to investigate the area of timed automata.

The passing of time is modelled as a global *tick* action, and a process counts time by synchronising with the global clock on action *tick*. Our framework deals very elegantly with the assumption of maximal progress, as well as with the basic correctness requirements for timed systems, related to zeno executions and time deadlock. The assumption of maximal progress states that time can only pass when no internal actions of the system are eligible. In other words, internal actions have priority over action *tick* [11]. We consider a system as *closed*, when the interesting part of the behaviour of its environment has been included in the model, and therefore the system is not expected to interact with components that are not part of it. In this case, all actions in the system can be considered as internal, and therefore they are *urgent* with respect to the passage of time. Therefore, in our approach, maximal progress for a closed system is imposed by making the tick action low priority: "`>> {tick}`". Additionally, the fact that a system is free of both zeno behaviours, and of time deadlocks, can be demonstrated by simply asserting the progress property: "`progress NoZeno_NoDeadlock = {tick}`" [11].

Two types of components in the BRP software architecture need to incorporate time, namely component type TIMER and component types R_CHANNEL and S_CHANNEL. Note that the channels for the receiver and sender need not be of different types. However, we provide a different model for channel R_CHANNEL; the messages that it transmits do not contain data, which results in R_CHANNEL having fewer states than S_CHANNEL. This reduces the state space of our case study.

A timer behaves as follows:

```
TIMER = ({tick, cancel} -> TIMER | tset[init:1..T2] -> COUNT_DOWN[init]),
COUNT_DOWN[time:1..T2] =  ( tick -> COUNT_DOWN[time-1]
                          | tset[init:1..T2] -> COUNT_DOWN[init]
                          | cancel -> TIMER),
COUNT_DOWN[0] =     ( {timeout, cancel} -> TIMER
                    | tset[init:1..T2] -> COUNT_DOWN[init]).
// no time can pass until timeout has been sorted out
```

A timer is a process which, when set (action *tset*) to a specific value *x*, counts *x* clock ticks and then produces a *timeout* event. The timer may be cancelled by the process that uses it, in which case it goes back to its initial state. When *x* clock ticks have been counted by the timer, i.e., when zero clock ticks remain to be counted (state *COUNT_DOWN[0]*), the timer does not allow time to pass because the process that has set it must handle the timeout immediately (immediately meaning before time passes, although other instantaneous events may happen in between, due to interleaving). Alternatively, if the process decides that it no longer needs the timer (action *cancel*), the latter returns to its initial state. Finally, the process may decide to reset the timer.

A lossy channel that may delay up to CD time units behaves as follows:

```
R_CHANNEL =  ( in -> (pick_del[dl:DELAYS] -> DECIDE[dl])
             | tick -> R_CHANNEL),
DECIDE[picked_value:DELAYS] =   if (picked_value == 0)
                                      then NO_DELAY
                                      else DELAY[picked_value],
DELAY[picked_value:DELAYS]=( when (picked_value>1) tick -> DELAY[picked_value-1]
                           | when (picked_value == 1) tick -> NO_DELAY
                           | in -> ERROR),
NO_DELAY =  ( {out, lose} -> R_CHANNEL | in -> ERROR).
```

The channel inputs a message, and picks a delay *dl* at random, within range DELAYS. It then counts *dl* clock ticks, before deciding whether the message is to be transmitted (action *out*) or lost (action *lose*). This channel has capacity of one, because we claim that this is sufficient when no premature timeouts occur (assumption (i)). To make sure that our claim is correct, we introduce the choice "`in -> ERROR`" at appropriate states of the channel. This means that, if a new message is input to the channel, while the channel has not finished with the transmission of the previous message, the channel transits to an ERROR state. If this state is reachable in the global LTS of BRP (detected by safety checks), then we will know that our claim is not correct.

It is important to allow the channel to pick delays at random, rather than setting its delay to the fixed value of CD. The channel delay is not standard; it is less than or equal to CD, which means that transmission times vary within some range of values. Indeed, this is a trap that we did not avoid in our initial model. As a result, the minimal value for timer T_3 was erroneously calculated to a smaller value.

The software architecture of BRP as illustrated in Figure 2 and Figure 3, describes the way components are put together, and has been used to produce the model of the entire protocol [7, 14]:

```
||BRP=(tx:TRANSMITTER || tr_chnl:S_CHANNEL || rx:RECEIVER || rec_chnl:R_CHANNEL)
      / {tx.send / tr_chnl.in,
         rx.rec / tr_chnl.out,
         rx.reply / rec_chnl.in,
         tx.ack / rec_chnl.out,
         accept/tx.accept,
         result/tx.result,
         deliver/rx.deliver,
         tick/ {tx.tick, rx.tick, rec_chnl.tick, tr_chnl.tick} } >> {tick}.
```

We give low priority to action tick (`>> {tick}`) in order to impose maximal progress. Maximal progress imposes a condition on timed models that time can only pass when no internal actions are enabled. This is a common assumption across a variety of models of timed concurrency, and restricts situations where an enabled internal action never occurs because all that keeps happening is the action *tick*, i.e. passage of time [18].

For the case of the BRP, we assume that both the S_CLIENT and R_CLIENT are always ready to accept output, and that S_CLIENT does not impose any timing constraints on the protocol to receive the next file. Finally, the correctness of the protocol does not depend on S_CLIENT being ready to provide a new file when the protocol is ready for it. As a result, we can model and check BRP as a *closed* system. We consider a system as closed when the interesting part of the behaviour of its environment has been included in the model, and therefore the system is not expected to interact with components that are not part of it. In a closed system, all actions may be viewed as internal, and therefore maximal progress reduces to giving low priority to action *tick*.

In the presence of maximal progress, one should be particularly careful with the specification of timed components. For example, there may be no meaning in modelling a choice between some action and the passage of time (*tick*); unless the action is blocked due to synchronisation reasons in the global model, the passage of time is not really eligible. This also justifies our model of the channel. The fact that transmission delay may vary for each message is modelled by action *pick_del* that "picks" a delay to be applied, rather than as a sequence of choices between actions *tick* and {*out*, *lose*}.

The models of the components that have not been discussed in this section can be found in the Appendix. We would like to make a remark here related to the way component RX has been modelled. According to the descriptions of the protocol provided in other papers [2, 5], the receiver is supposed to reset its local timer only when the message it receives is a new message. In our model of RX, RX resets its local timer each time it receives a message from the sender, whether that is a duplicate or not. In essence, this does not affect the correctness of protocol. However, the minimum value for timer T_2 is smaller in our case, which also results in our models to have fewer states (since they need to count to a smaller value). This is not, of course, supposed to suggest the way in which the protocol is to be implemented, which should be left to the developer of a system in which BRP may be applicable.

4.3 Properties

Our approach supports the following ways of introducing "properties" to be checked on a model.

A. Include ERROR states in component specifications.

Our tools detect safety violations by checking reachability of the error state in the LTS of a system [9]. The error state is state "-1" in our LTSs, and is represented by the pre-defined process

ERROR in FSP specifications. Error states can be introduced either explicitly in the FSP models of the system components, or by means of properties that are composed with the system. In the former case, the system designer may write (…*a* -> ERROR…) to state that at action *a* takes the component from its current state to an illegal state. Presence of the error state in the final model reflects the fact that the illegal state of the component is not avoided in all cases.

For example, in Section 3.2 we mentioned that at the receiver's end, if the last frame of a file is received, the next new frame must be the first frame of the following file. We have included this assumption in the behaviour of RX, as follows:

```
...
UNKNOWN_AB = ( rec[ab:BIT][f:BIT][l:BIT][x:CHUNKS] ->
                          if (f != 1) then ERROR //expecting first chunk...
                          else NEW_MESG[ab][f][l][x]),
...
KNOWN_AB[eab:BIT][got_last:BOOL] =
             ( rec[eab][f:BIT][l:BIT][x:CHUNKS] ->
                          if ((f!=1) && (got_last == 'true)) then ERROR
                          else NEW_MESG[eab][f][l][x]),
...
```

When RX is in a state where the previous file transmission has been completed or aborted, RX transits in an error state if it receives a new chunk that is not the first one of a file.

B. Introduce component properties that express assumptions on the environment.

Our approach to model checking allows us to introduce properties at any layer of the structural hierarchy of a system. Properties are expressed as automata, that are included into the system as ordinary components, and that monitor the behaviour of the system without affecting it, unless the latter violates the property that they express [14]. Local properties that are introduced at intermediate levels of the hierarchy normally refer to properties that apply to subcomponents (and that are checked at the component level), or to assumptions that the component makes on its context, and that justify its behaviour. In the latter case, the component will not usually satisfy the properties in isolation, but rather when composed with part of, or the whole of its context.

An example of such a property is the one that we described in Section 3.2, which refers to component RX, and has to do with synchronisation on the alternating-bit scheme: "*at the receiver's end, any consecutive messages tagged with the same bit bear identical contents*". This property is expressed as follows:

```
property ALT_BIT = INIT_MODE,
INIT_MODE = ( rec[ab:BIT][f:BIT][l:BIT][x:CHUNKS] -> CHECK_MODE[ab][f][l][x]),
CHECK_MODE[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
       ( rec[ab][f][l][x] -> CHECK_MODE[ab][f][l][x] // same ab => same contents
       | rec[!ab][f:BIT][l:BIT][y:CHUNKS] -> CHECK_MODE[!ab][f][l][y]
       | timeout -> INIT_MODE).

||RECEIVER = (RX || TIMER || ALT_BIT).
```

The receiver process is thus made up of component RX combined with its local property and a timer.

C. Introduce properties that express the system requirements.

At this stage, we introduce properties that express the requirements we have stated for the system in Section 2. The requirements on message delivery to R_CLIENT are all expressed by the following property, whose corresponding LTS is illustrated in Figure 5:

```
property REC_RESULT = CHECK_DELIVERY[1],
CHECK_DELIVERY[x:1..FILE_LENGTH] =
            if (x == 1 && FILE_LENGTH == 1)
                  then ( deliver[1].ok -> REC_RESULT)
            else if (x == 1 && FILE_LENGTH >1)
                  then ( deliver[1]['first] -> CHECK_DELIVERY[2])
            else if (x > 1 && x < FILE_LENGTH)
                  then ( deliver.nok -> REC_RESULT  // transfer aborted
                       | deliver[x]['int] -> CHECK_DELIVERY[x+1])
            else if (x == FILE_LENGTH)
                  then ( deliver.nok -> REC_RESULT  // transfer aborted
                       | deliver[FILE_LENGTH].ok -> REC_RESULT).
```

The above property checks that for a file of length FILE_LENGTH, chunks are delivered in order and with the correct indication, until either transmission is aborted in which case indication nok is delivered, or the last chunk (id FILE_LENGTH) is delivered with indication ok. Moreover, whenever at least one chunk of a file is received, an indication ok or nok is delivered, before a new file transmission begins. Finally, each chunk is delivered at most once.

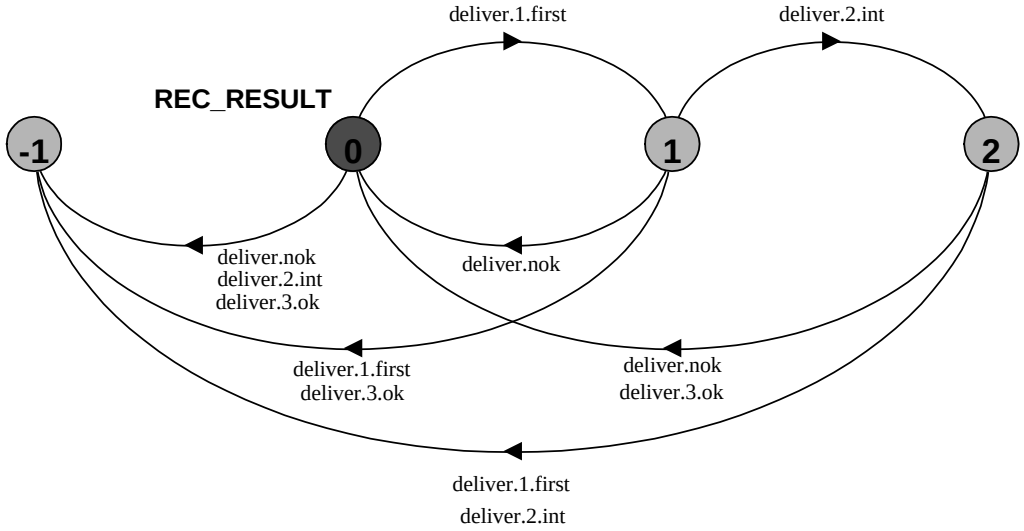

Figure 5: LTS for safety property REC_RESULT

The requirements at the sending client's side are expressed by the following property, whose corresponding LTS is illustrated in Figure 6:

```
property SEND_RESULT = ( accept.file -> DECIDE),
DECIDE = ( deliver[FILE_LENGTH].ok -> {result.ok, result.dknow} -> SEND_RESULT
         | {result.dknow, result.nok} -> SEND_RESULT).
```

The property states that, after a file is accepted, one of the following outcomes are legal. Only if the last chunk is received by R_CLIENT, can the result reported be *ok*. This covers the requirement according to which if S_CLIENT gets an indication that transfer of a file was successful, then all chunks must have been delivered. It suffices of course to check that the last one is delivered, since we are also checking with property REC_RESULT that chunks are delivered in order. Moreover, the result can only be *nok* if the last chunk has not been transmitted. This expresses the requirement that at least one chunk has not been transmitted if the result reported is *nok*.

The above safety requirements are checked by composing the model of the BRP with the properties, as follows:

```
|| SAFE_BRP = (BRP || REC_RESULT || SEND_RESULT).
```

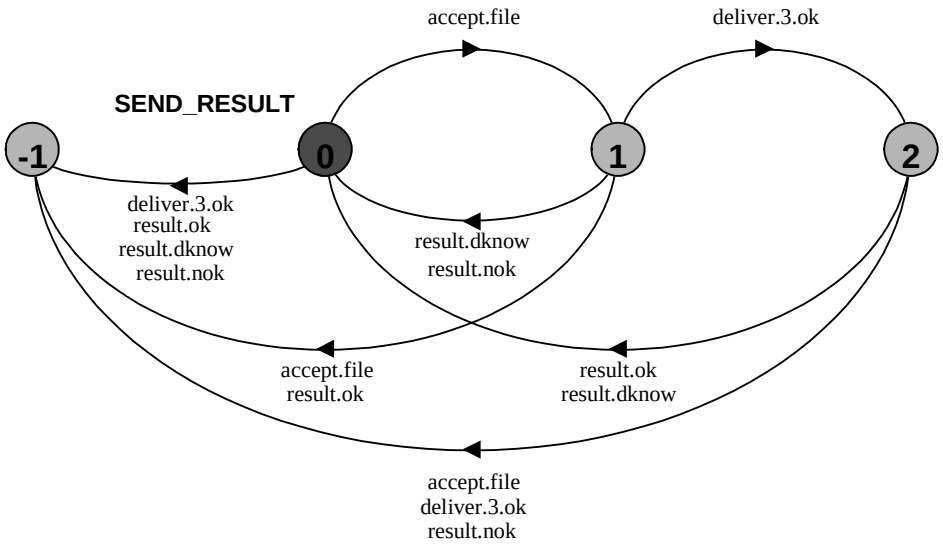

Figure 6: LTS for safety property SEND_RESULT

Finally, we also need to introduce some liveness requirements. Firstly, the basic requirement for timed systems is that no time deadlock occurs, and no zeno executions are exhibited. Both properties are expressed by the following progress property [11]:

```
progress NO_ZENO_NO_HALT = {tick}
```

Naturally, if action *tick* can always eventually happen, neither time deadlock, nor zeno executions (where infinitely many actions occur without time passing) are possible.

Another progress property that applies to our case is that files can always eventually be accepted, meaning that the protocol can always conclude with the transmission of a file and be ready to accept another:

```
progress ACCEPT = {accept.file}
```

As we expect property ACCEPT to hold, we can express property $\Box(accept \Rightarrow \Diamond\ result)$ as follows [19]:

```
progress RESULT = if {accept.file} then {result.ok, result.nok, result.dknow}
```

The above, combined with safety property SEND_RESULT, guarantees that for each file accepted, a result is eventually produced. This completes modelling of the requirements set for the protocol in Section 2.

5 Analysis

In order to check the correctness of the protocol, we perform compositional reachability analysis (CRA) as described in [14]. That means that the model for BRP is composed from those of its components gradually, as described by its corresponding FSP expression (the latter has been derived by the software architecture descriptions of Figure 2 and Figure 3. When analysing timed systems with maximal progress, one should be particularly careful about applying minimisation with respect to observational equivalence at intermediate stages of analysis. The reason is that cycles made up of internal actions may disappear. When a choice between an internal action that starts such a cycle with a *tick* action exists, different results will occur by applying maximal progress to the LTS before and after minimisation. In the former case, *tick* will never be selected. In the latter, the cycle will disappear with minimisation, and therefore *tick* will be the only choice at the corresponding state. As we have not yet determined how we would like to deal with this problem, we do not apply minimisation at intermediate phases of CRA. Rather, after checking that the protocol is correct, we apply minimisation to obtain a smaller LTS that describes the behaviour of the protocol in terms of its interface, and can be used as a component of a larger system.

5.1 Untimed version

To show that the correctness of the protocol depends on the timing assumptions that we have set, we check the protocol without taking time into account. We perform this by removing the TIMER components from both the RECEIVER and the TRANSMITTER. As a result, timeouts in TX and RX occur non-deterministically since they no longer need to synchronise with the TIMERS. We use LTSA to check BRP against safety, and get the following result:

```
States Composed: 55340 Transitions: 194563 in 16013ms
Trace to property violation in tr_chnl:S_CHANNEL:
             accept.file
             tx.send.0.1.0.1
             tx.send.0.1.0.1
```

In the above counterexample, the first chunk is sent twice. However, as indicated by the fact that the error state is derived from component S_CHANNEL, the channel has not dealt with the first chunk, when it receives a new copy of it. That indicates that TX has timed out prematurely. This violation therefore describes the fact that channels of capacity one are not sufficient when TX and RX may timeout prematurely. This, however, does not mean that the protocol requires infinite capacity channels. In fact, since message loss may happen anyway, the correctness of the protocol should not be affected by the fact that a channel has capacity one, and therefore loses messages that are sent to it while it is full. For this reason, we remove the transitions to ERROR states from the channel specifications, and check BRP again. The following result is obtained:

```
States Composed: 55340 Transitions: 184521 in 15542ms
Trace to property violation in rx:RECEIVER:
             accept.file
             tx.send.0.1.0.1
```

```
            rx.rec.0.1.0.1
            deliver.1.first
            rx.reply
            tx.ack
            tx.send.1.0.0.2
            deliver.nok
            rx.rec.1.0.0.2
```

This counterexample describes a scenario where the RX times out, assuming that the file transmission has been aborted (obvious from the fact that it reports *deliver.nok*). It then expects that the new chunk that will be received will be the first chunk of a new file, whereas the chunk that is received is in fact the second chunk of the first file (*rx.rec.1.0.0.2*). That indicates that TX has not in fact aborted transmission, but the receiver has simply timed out too early. The error state roots from the specification of component RX, intended to detect exactly such problems.

If, in addition, we check component CHECK_BRP, which includes the safety properties that express the requirements, we get as a result that these properties are also violated:

```
property SEND_RESULT violation..
property REC_RESULT violation...
```

We conclude that, indeed, the protocol depends on its timing constraints for correctness. In what follows, we analyse the protocol again given its original specifications, i.e. including both the TIMER components and the error states in the channel specifications.

5.2 Timed version

With the TIMER components included, and with the values calculated in Section 3.3, we check first the BRP component, and subsequently SAFE_BRP, against safety. The following results are obtained:

```
States Composed: 5029 Transitions: 9002 in 260ms // result for BRP
No deadlocks/errors

States Composed: 5050 Transitions: 9027 in 211ms // result for SAFE_BRP
No deadlocks/errors
```

In both cases, the state space is significantly smaller than that obtained for the untimed version, since the timing constraints also restrict the behaviour of the protocol, as required for correctness. We thus prove that the protocol satisfies all its requirements given the timing constraints that we have imposed. Additionally, our results show that the channels never need to receive a new message while another message is in transit, and therefore capacity of one is sufficient. Note that the increase in the state-space introduced by composing BRP with its properties in SAFE_BRP is negligible. Moreover, when properties are violated, as in the following case, the state-space may even be reduced, because properties prune out behaviour that roots at erroneous system states [9].

An interesting question at this point is: are the values T_1, T_2 and T_3, calculated in Section 3.3 minimal? Or would the protocol still work with smaller values? Let us try to reduce value T_2 by one, and check component BRP for safety:

```
States Composed: 4971 Transitions: 8955 in 381ms
Trace to property violation in rx:RECEIVER:
            accept.file
            tx.send.0.1.0.1
```

```
rx.rec.0.1.0.1
deliver.1.first
rx.reply
(tick) * 2  // abbreviation for two consecutive ticks
tx.ack
tx.send.1.0.0.2
(tick) * 5
tx.send.1.0.0.2
(tick) * 2
deliver.nok
rx.rec.1.0.0.2
```

Similarly to the counterexample obtained in Section 4.1, this scenario describes an execution where RX timeouts too early. As a result, it receives the second chunk of the previous file when it expects the first chunk of a new file. We set T_2 back to its original value, and now try decrementing the value of T_3. We obtain the following result:

```
property rx:RECEIVER violation......
States Composed: 5194 Transitions: 9388 in 381ms
Trace to property violation in ?:
       accept.file
       tx.send.0.1.0.1
       rx.rec.0.1.0.1
       deliver.1.first
       rx.reply
       tx.ack
       tx.send.1.0.0.2
       rx.rec.1.0.0.2
       deliver.2.int
       rx.reply
       tx.ack
       tx.send.0.0.1.3
       (tick) * 5
       tx.send.0.0.1.3
       (tick) * 2
       rx.rec.0.0.1.3
       deliver.3.ok
       rx.reply
       (tick) * 3
       result.dknow
       (tick) * 7
       accept.file
       tx.send.0.1.0.1
       rx.rec.0.1.0.1
```

This counterexample is much longer, and takes a bit more effort to understand. The tool is not able to locate the component where the ERROR state roots. However, by careful study of the counterexample, we understand that the alternating-bit scheme is broken, and therefore property ALT_BIT that has been included in the RECEIVER is violated. We describe the problem in the last 9 lines of the counterexample. The receiver gets a message *rx.rec.0.0.1.3*, replies to it, and is waiting either for a new message to arrive, or for a timeout to occur. The timeout will occur after T_2 = 10 time units. In the meanwhile, the TRANSMITTER does not receive an acknowledgement

for this last chunk of the file. As it has completed all its retransmissions, it aborts transmission, reports *result.dknow*, and after T_3 time units, accepts a new file to transmit. It also restarts the alternating-bit scheme with value 0. Unfortunately, the RECEIVER has not yet timed out, and therefore it expects that messages tagged with value 0 are retransmissions. Property ALT_BIT detects the fact that a message with different content (*rx.rec.0.1.0.1*) is assumed as a retransmission, and leads BRP to an ERROR state.

We can see that, long counterexamples are difficult to understand. This is particularly so in cases where time is involved, as above; counterexamples become longer, and more complicated. We therefore think that in such cases, it would be useful to be able to replay counterexamples in the context of some domain-specific animation, where such scenarios take shape, and are much easier to interpret. We intend, in the future, to implement a domain-specific animation for the protocol, as described in [17].

We would like to mention here that we have checked the protocol for several values of both our timer constants, and other constants such as FILE_LENGTH, TRIES, etc. When values of timer constants are increased, the correctness of the protocol remains unaffected, in contrast with cases where they are decreased, as already discussed. We are therefore confident that the values we have calculated are indeed minimal.

5.3 Liveness

We have checked the safe version of the protocol against the liveness properties introduced in Section 4.3. No violations were detected.

5.4 Abstraction

Having proved correctness of the BRP, we can now obtain a model of its interface behaviour, as follows:

```
minimal || UNTIMED_BRP = BRP \ {tick}.
minimal || ABSTRACTED_BRP = UNTIMED_BRP @ {accept, result, deliver}.
```

The prefix *minimal* describes the fact that we wish the LTS for a specific component to be minimised with respect to observational equivalence. We perform minimisation in two steps. Firstly, we abstract time from our model, since it is no longer needed. Second, we abstract all actions that do not form part of the component's interface (see Figure 2). Minimisation in this fashion is more efficient, and this is the way we proceed for timed case studies. This is due to the way in which observational minimisation is implemented. Observational minimisation is applied in two steps [20]. First, a transformation is performed to the graph of the system: the reflexive transitive closure of the τ relation is computed, and observable actions are made to absorb internal ones. Second, strong minimisation is applied to the graph resulting from the first step. When the τ relation is very large, the graph obtained from the first stage is very large, and therefore expensive to minimise.

The LTS of ABSTRACTED_BRP has just 29 states. This LTS can be used in the context of larger systems. Abstraction thus allows one to avoid including unnecessary details, as one moves higher on the architectural hierarchy of a system, thus often avoiding state explosion as well.

If we restrict the file length to simply one chunk, we obtain a very small LTS, which can be clearly illustrated, and gives some intuition about the way the protocol works (see Figure 7). The

diagram clearly illustrates that non-determinism is introduced (at states 1 and 3) by the fact that unreliable channels are used for the transmission.

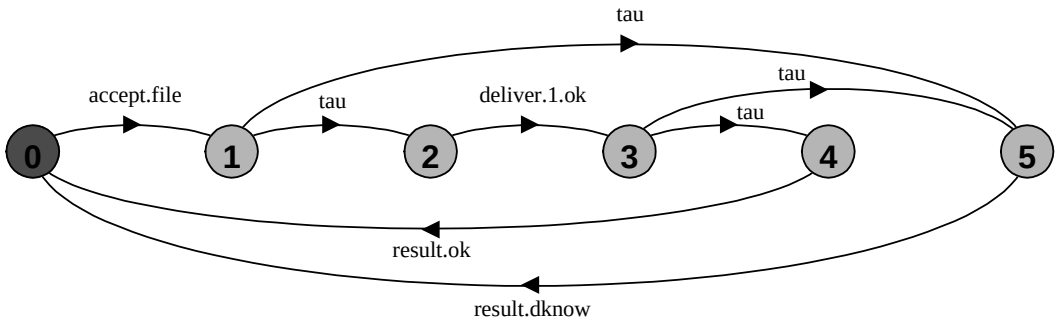

Figure 7: Abstracted BRP behaviour for FILE_LENGTH = 1

6 Discussion and Conclusions

This paper reported the experience obtained by applying our approach to the modelling and analysis of the BRP. Our approach uses CRA based on the software architecture to incrementally generate, minimise and analyse the behaviour of a system. It is supported by the LTSA, a tool written in Java™ that can be run as an application or applet, and which is available at http://www-dse.doc.ic.ac.uk/concurrency. Correctness of the BRP relies on specific timing assumptions, an aspect that cannot be made explicit in an un-timed model of the protocol. Although the LTSA tool does not contain features designed specifically for dealing with real-time systems, we have shown that it handles discrete time systems in an efficient way. In particular, our notions of fairness, progress, and action priority, proved elegant in modelling maximal progress and the basic correctness requirements in timed systems, i.e. that a system exhibits no zeno executions and no time deadlocks.

Discrete time counted in terms of clock ticks is a straightforward way of modelling timed systems in tools that are not specifically aimed at such systems. It is also easy to understand, since it does not require additions to the specification language of a tool. However, it can be error-prone and tedious. As discussed in the paper, one needs to have a clear understanding of the timed guards that control a transition, and must implement those in an appropriate fashion. Subtle mistakes can easily be committed in this context, as was the case for the channel delay in section 4.2, for example. These modelling concerns are even bigger in the presence of the maximal progress assumption.

Modelling timed systems becomes much easier with timed automata, although one has to first invest the time to understand the new model [21]. The abstraction mechanisms associated with timed automata can also be exploited to avoid the significant increase in the state-space of a system, which can result from selecting an unnecessarily fine unit of time in models that count time explicitly. We intend to investigate the work on timed automata further, and we consider the possibility of extending the LTSA in that direction, although in a way that keeps in line with our existing approach and requirements.

In analysing timed systems in the context of CRA, we currently avoid minimisation at intermediate subsystems, and perform minimisation only after the global system has been obtained. Moreover, we apply maximal progress only at the global system. Minimisation at the final stage

provides useful abstractions that focus on specific aspects of the system. Additionally, when the system is used as a component of a larger system that is not concerned with timing aspects, the abstracted model can be used in order to avoid state explosion. However, an issue that we intend to investigate in the future is how the notion of time and maximal progress can be incorporated more efficiently in our compositional setting, in particular as far as compositional minimisation is concerned.

Acknowledgements

I would like to acknowledge my colleagues Jeff Kramer and Jeff Magee for discussions on timed systems in the context of the LTSA tool. The early model of the BRP was discussed with Pedro D'Argenio from the University of Twente, whose comments were invaluable in obtaining the version presented in this paper. My visit at the University of Twente was organised by Prof. Ed Brinksma. I gratefully acknowledge the EPSRC (BEADS GR/M 24493) and the EU (C3DS ESPRIT project 24962) for their financial support.

References

[1] Helmink, L., Sellink, M.P.A., and Vaandrager, F.W. "Proof-Checking a Data Link Protocol", in *Proc. of the 1st International Workshop on Types for Proofs and Programs (TYPES '93)*. May 1993, Nijmegen, The Netherlands. Springer-Verlag, Lecture Notes in Computer Science 806, pp. 127-165. H.P. Barendregt and T. Nipkow, Eds.

[2] Groote, J.F. and Pol, J.v.d. "A Bounded Retransmission Protocol for Large Data Packets", in *Proc. of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96)*. July 1996, Munich, Germany. Springer-Verlag, Lecture Notes in Computer Science 1101, pp. 536-550. M. Wirsing and M. Nivat, Eds.

[3] Havelund, K. and Shankar, N. "Experiments in Theorem Proving and Model Checking for Protocol Verification", in *Proc. of the Third International Symposium of Formal Methods Europe (FME'96)*. March 1996, Oxford, UK. Springer-Verlag, Lecture Notes in Computer Science 1051, pp. 662-681. M.-C. Glaudel and J. Woodcock, Eds.

[4] Mateescu, R. "Formal Description and Analysis of a Bounded Retransmission Protocol", in *Proc. of the International Workshop on Applied Formal Methods in System Design (COST 247)*. June 1996, University of Maribor, Slovenia.

[5] D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., and Tretmans, J. "The Bounded Retransmission Protocol must be on time!", in *Proc. of the 3d International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*. 1997, Enschede, The Netherlands. Springer-Verlag, Lecture Notes in Computer Science 1217, pp. 416-431. E. Brinksma, Ed.

[6] Giannakopoulou, D., Kramer, J., and Cheung, S.C., *Analysing the Behaviour of Distributed Systems using Tracta.* Journal of Automated Software Engineering, special issue on Automated Analysis of Software, Vol. **6**(1), January 1999: pp. 7-35.

[7] Magee, J., Kramer, J., and Giannakopoulou, D. "Behaviour Analysis of Software Architectures", in *Proc. of the 1st Working IFIP Conference on Software Architecture (WICSA1)*. 22-24 February 1999, San Antonio, TX, USA.

[8] Magee, J., Kramer, J., and Giannakopoulou, D. "Analysing the Behaviour of Distributed Software Architectures: a Case Study", in *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*. October 1997, Tunis, Tunisia, pp. 240-245.

[9] Cheung, S.C. and Kramer, J., *Checking Safety Properties Using Compositional Reachability Analysis.* ACM Transactions on Software Engineering and Methodology, Vol. **8**(1), January 1999: pp. 49-78.

[10] Cheung, S.C., Giannakopoulou, D., and Kramer, J. "Verification of Liveness Properties using Compositional Reachability Analysis", in *Proc. of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*. September 1997, Zurich, Switzerland. Springer, Lecture Notes in Computer Science 1301, pp. 227-243. M. Jazayeri and H. Schauer, Eds.

[11] Giannakopoulou, D., Magee, J., and Kramer, J. "Checking Progress with Action Priority: Is it Fair?", in *Proc. of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*. September 1999, Toulouse, France. Springer, Lecture Notes in Computer Science 1687, pp. 511-527. O. Nierstrasz and M. Lemoine, Eds.

[12] Magee, J. and Kramer, J., *Concurrency: State Models & Java Programs*: John Wiley & Sons, 1999.

[13] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. "Specifying Distributed Software Architecture", in *Proc. of the 5th European Software Engineering Conference (ESEC'95)*. September 1995, Sitges, Spain, Lecture Notes in Computer Science 989, pp. 137-153. W. Schäfer and P. Botella, Eds.

[14] Giannakopoulou, D., "Model Checking for Concurrent Software Architectures", PhD Thesis, March 1999.

[15] Magee, J., Kramer, J., and Giannakopoulou, D. "Software Architecture Directed Behaviour Analysis", in *Proc. of the Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9)*. April 16-18 1998, Ise-shima, Japan, pp. 144-146.

[16] Wolper, P. "Expressing Interesting Properties of Programs in Propositional Temporal Logic", in *Proc. of the 13th ACM Symposium on Principles of Programming Languages*. 13-15 January 1986, St. Petersburg Beach, Florida. Association for Computing Machinery, pp. 184-193.

[17] Magee, J., Pryce, N., Giannakopoulou, D., and Kramer, J. "Graphical Animation of Behavior Models", in *Proc. of the 22d International Conference on Software Engineering (ICSE' 2000)*. June 2000, Limerick, Ireland.

[18] Roscoe, A.W., *The Theory and Practice of Concurrency*: Prentice Hall, 1998.

[19] Giannakopoulou, D., Magee, J., and Kramer, J., "Fairness and Priority in Progress Property Analysis", Department of Computing, Imperial College of Science, Technology and Medicine, Research Report, DoC 99/3, December 1999.

[20] Kanellakis, P.C. and Smolka, S.A., *CCS Expressions, Finite State Processes, and Three Problems of Equivalence*. Information and Computation, Vol. **86**(1), May 1990: pp. 43-68.

[21] Alur, R. and Dill, D.L., *A Theory of Timed Automata.* Theoretical Computer Science, Vol. **126**, 1994: pp. 183-235.

APPENDIX

```
/****************** FSP code for the BRP case study ******************/

range BIT = 0..1
const TRIES = 2
const FILE_LENGTH = 3
const CD = 2  // max channel delay
const MAX_WAIT = 3

range CHUNKS = 1..FILE_LENGTH
range DELAYS = 0..CD

set LABELS = {first, int, ok, nok}
set BOOL = {true, false}

const T1 = 2*CD + 1
const T2 = T1 * TRIES
const T3 = T2 - T1 + CD + 1


TX = ACCEPT[0],
ACCEPT[ab:BIT] = ( accept.file -> oper.reset -> if (FILE_LENGTH == 1)
                                                  then SEND[ab][1][1][1]
                                                  else SEND[ab][1][0][1]
                 ),
SEND[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
             ( oper.inc -> send[ab][f][l][x] -> tset[T1] -> SENDING[ab][f][l][x]
             | full -> DECIDE[l]),
DECIDE[l:BIT] = ( when (l ==1) result.dknow -> SYNCHRONISE
                | when (l == 0) result.nok -> SYNCHRONISE),
SYNCHRONISE = (tset[T3] -> timeout -> ACCEPT[0]),
SENDING[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
             ( timeout -> SEND[ab][f][l][x]
             | ack -> cancel -> COMPLETED[ab][l][x]),
COMPLETED[ab:BIT][l:BIT][x:CHUNKS] =
             ( when (l == 1) result.ok -> ACCEPT[!ab]
             | when (l == 0) oper.reset -> if (x+1 == FILE_LENGTH)
                                           then SEND[!ab][0][1][x+1]
                                           else SEND[!ab][0][0][x+1]
)  + {tset[i:1..T2]}. // timer operations


// variable used by the transmitter to count the total transmissions of a chunk
CNT = COUNTER[0],
COUNTER[i:0..TRIES-1] = (oper.inc ->COUNTER[i+1] | oper.reset ->COUNTER[0]),
COUNTER[TRIES] = (full -> COUNTER[TRIES] | oper.reset->COUNTER[0]).
```

```
TIMER = ({tick, cancel} -> TIMER | tset[init:1..T2] -> COUNT_DOWN[init]),
COUNT_DOWN[time:1..T2] =  ( tick -> COUNT_DOWN[time-1]
                          | tset[init:1..T2] -> COUNT_DOWN[init]
                          | cancel -> TIMER),
COUNT_DOWN[0] =     ({timeout, cancel} -> TIMER
                    | tset[init:1..T2] -> COUNT_DOWN[init]).
// no time can pass until timeout has been sorted out

||TRANSMITTER = (TX || CNT || TIMER) @ {send, accept, result, ack, tick}.

S_CHANNEL =  ( in[ab:BIT][f:BIT][l:BIT][x:CHUNKS] ->
                          (pick_del[dl:DELAYS] -> DECIDE[ab][f][l][x][dl])
             | tick -> S_CHANNEL),
DECIDE[ab:BIT][f:BIT][l:BIT][x:CHUNKS][delay:DELAYS] =
             if (delay == 0) then NO_DELAY[ab][f][l][x]
                             else DELAY[ab][f][l][x][delay],
NO_DELAY[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
             ( {out[ab][f][l][x], lose} -> S_CHANNEL
             | in[ab:BIT][f:BIT][l:BIT][x:CHUNKS] -> ERROR),
DELAY[ab:BIT][f:BIT][l:BIT][x:CHUNKS][delay:0..CD] =
             ( when (delay > 1) tick ->  DELAY[ab][f][l][x][delay-1]
             | when (delay == 1) tick -> NO_DELAY[ab][f][l][x]
             | in[ab:BIT][f:BIT][l:BIT][x:CHUNKS] -> ERROR
             ) @ {in, out, tick}.

R_CHANNEL =  ( in -> (pick_del[dl:DELAYS] -> DECIDE[dl])
             | tick -> R_CHANNEL),
DECIDE[picked_value:DELAYS] =    if (picked_value == 0)
                                 then NO_DELAY
                                 else DELAY[picked_value],
NO_DELAY =  ( {out, lose} -> R_CHANNEL | in -> ERROR),
DELAY[picked_value:DELAYS] =
             ( when (picked_value > 1) tick ->  DELAY[picked_value-1]
             | when (picked_value == 1) tick -> NO_DELAY
             | in -> ERROR) @ {in, out, tick}.

RX = UNKNOWN_AB,
UNKNOWN_AB = ( rec[ab:BIT][f:BIT][l:BIT][x:CHUNKS] ->
                    if (f != 1)  then ERROR //expecting first chunk...
                                 else NEW_MESG[ab][f][l][x]),
KNOWN_AB[eab:BIT][got_last:BOOL] =
(rec[eab][f:BIT][l:BIT][x:CHUNKS] -> if ((f!=1) && (got_last == 'true))
                                       then ERROR
                                       else NEW_MESG[eab][f][l][x]
|rec[!eab][f:BIT][l:BIT][x:CHUNKS]-> reply-> tset[T2] -> KNOWN_AB[eab][got_last]
| timeout -> if (got_last == 'false) then (deliver['nok] -> UNKNOWN_AB)
                                    else UNKNOWN_AB),
NEW_MESG[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
             if (f == 1 && l ==0) then DELIVER[ab]['first]['false][x]
             else if (l==1) then DELIVER[ab]['ok]['true][x]
             else DELIVER[ab]['int]['false][x],
DELIVER[ab:BIT][lb:LABELS][got_last:BOOL][x:CHUNKS] =
             (deliver[x][lb] -> reply -> tset[T2] -> KNOWN_AB[!ab][got_last])
```

```
+ {tset[i:1..T2], cancel}.
// property ALT_BIT
// consecutive messages with the same alt-bit must have identical contents
// unless a timeout comes between them

property ALT_BIT = INIT_MODE,
INIT_MODE = ( rec[ab:BIT][f:BIT][l:BIT][x:CHUNKS] -> CHECK_MODE[ab][f][l][x]),
CHECK_MODE[ab:BIT][f:BIT][l:BIT][x:CHUNKS] =
             ( rec[ab][f][l][x] -> CHECK_MODE[ab][f][l][x]
             | rec[!ab][f:BIT][l:BIT][y:CHUNKS] -> CHECK_MODE[!ab][f][l][y]
             | timeout -> INIT_MODE).

||RECEIVER = (RX || TIMER || ALT_BIT) @ {reply, rec, deliver, tick}.

||BRP=(tx:TRANSMITTER || tr_chnl:S_CHANNEL || rx:RECEIVER || rec_chnl:R_CHANNEL)
       / {tx.send / tr_chnl.in,
          rx.rec / tr_chnl.out,
          rx.reply / rec_chnl.in,
          tx.ack / rec_chnl.out,
          accept/tx.accept,
          result/tx.result,
          deliver/rx.deliver,
          tick/ {tx.tick, rx.tick, rec_chnl.tick, tr_chnl.tick} } >> {tick}.


/************************** SAFETY ******************************/

property REC_RESULT = CHECK_DELIVERY[1],
CHECK_DELIVERY[x:1..FILE_LENGTH] =
             if (x == 1 && FILE_LENGTH == 1)
                    then ( deliver[1].ok -> REC_RESULT)
             else if (x == 1 && FILE_LENGTH >1)
                    then ( deliver[1]['first] -> CHECK_DELIVERY[2])
             else if (x > 1 && x < FILE_LENGTH)
                    then ( deliver.nok -> REC_RESULT  // transfer aborted
                         | deliver[x]['int] -> CHECK_DELIVERY[x+1])
             else if (x == FILE_LENGTH)
                    then ( deliver.nok -> REC_RESULT  // transfer aborted
                         | deliver[FILE_LENGTH].ok -> REC_RESULT).

property SEND_RESULT = ( accept.file -> DECIDE),
DECIDE = ( deliver[FILE_LENGTH].ok -> {result.ok, result.dknow} -> SEND_RESULT
        | {result.dknow, result.nok} -> SEND_RESULT).

|| SAFE_BRP = (BRP || REC_RESULT || SEND_RESULT).

/************************** LIVENESS ******************************/

progress NO_ZENO_NO_HALT = {tick}
progress ACCEPT = {accept.file}
progress RESULT = if {accept.file} then {result.ok, result.nok, result.dknow}
// ACCEPT and RESULT guarantee [](accept => <> result)
```