An Event-B Model for On-Demand Streaming

Sandvik, Petter; Sere, Kaisa; Walden, Marina

Published: 01/01/2010

Document Version
Final published version

Document License
Unknown

Petter Sandvik | Kaisa Sere | Marina Waldén

An Event-B Model for On-Demand Streaming

Turku Centre *for* Computer Science

TUCS Technical Report
No 994, December 2010

An Event-B Model for On-Demand Streaming

Petter Sandvik
Department of Information Technologies, Åbo Akademi University
Turku Centre for Computer Science
Joukahaisenkatu 3–5, 20520 Turku, Finland
petter.sandvik@abo.fi

Kaisa Sere
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3–5, 20520 Turku, Finland
kaisa.sere@abo.fi

Marina Waldén
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3–5, 20520 Turku, Finland
marina.walden@abo.fi

TUCS Technical Report
No 994, December 2010

Abstract

As services and applications move away from the one-to-may relationship of the client-server model towards many-to-many relations such as cloud-based services and peer-to-peer networks, there is a need for a reusable model of how a node could work in such a network that transfers content between nodes. While distributed systems and in particular peer-to-peer technology has found success in file transfer use, on-demand media streaming has so far largely eluded this kind of decentralisation. In this paper we use the Event-B formalism to describe a structured formal model, from the more general node in a content transfer network to the more specific one of a peer-to-peer file sharing based content streaming.

Keywords: Formal Modelling, Streaming, Event-B, Rodin Platform Tool, Refinement

TUCS Laboratory
Distributed Systems

1 Introduction

There is a trend in moving services and applications away from the one-to-many relationship of the traditional client-server model towards many-to-many relations such as cloud-based services and peer-to-peer networks. These latter two form a part of the "utility computing vision" [6] in which computer services are accessed without needing to know the specific underlying structure. Because of this, there is a need for a model of how a reliable node could work in such a network that transfers content between nodes. We have created a formal model of this kind of node, in such a way that our model can be reused and adapted to many different applications.

For large-scale data transfer, such as installation media for operating systems and data files for multiplayer games, peer-to-peer file sharing technologies such as BitTorrent have become important. However, the out-of-order nature of BitTorrent file transfers have made them unsuitable for streaming media applications. It has already been argued [26] that with some modifications, it would be possible to create a streaming media solution based on BitTorrent. Indeed, there are proprietary and commercial efforts to create exactly such a thing, for instance "to turn BitTorrent into a point-click-watch experience much more similar to YouTube" [25], and a hybrid web/peer-to-peer video solution based on technology developed by the P2P-Next consortium [19] has been employed on Wikipedia [5].

We have previously described our idea [21] as well as presented an approach [22] for modifying the BitTorrent piece selection to work in an on-demand content streaming situation. In this paper we describe a structured formal model, from the more general node in a content transfer network to the more specific one of our approach, using the Event-B [2] formalism. We will describe this formalism in section 2, and in Section 3 we will describe on-demand streaming and the technologies we base our work on. In Section 4 we show the steps we take to create a formal model of our application using Event-B. Section 5 concludes this report with discussion about our results and future work.

2 Event-B

Event-B is a formalism based on Action Systems [4, 27] and the B Method [1]. In Event-B development is carried out stepwise from abstract specification to concrete implementation. In order to achieve a reliable system we use superposition refinement [3, 15] to add functionality while preserving the overall consistency, which means that we add new variables and functionality in such a way that it prevents the old functionality from being disturbed [23]. In order to prove the correctness of each step of the development, we rely on the Rodin Platform [10] tool where proof obligations are generated automatically and proven either auto-

matically or interactively. This integrated tool support is a major reason for us to choose Event-B as the formalism to use for a model of this kind.

Each model in Event-B consists of two parts. The first part, called a *context*, describes the static parts such as constants. The properties of the constants are written as *axioms*. The second part is called a *machine*, and it contains the dynamic parts such as variables and events. Properties that should be preserved during execution are written as a list of *invariants*. If the machine is a refinement of another, more abstract machine, the keyword *refines* and the name of the previous machine is included. Figure 1 shows the structure of an Event-B machine.

```
machine machine-name
  refines abstract-machine
  sees context-name
  variables list of variables
  invariants list of invariants/predicates
  events
    event INITIALISATION
      then
        actions
    end
    event other_event
      ...
    end
    ...
end
```

Figure 1: *A machine in Event-B. Based on Abrial [2].*

The `INITIALISATION` event is executed only once, at the very beginning. As the name implies, this event is used to set the initial values of the variables. Other events follow the structure shown in Figure 2.

The keyword *refines* is used whenever an event is a refinement of a more abstract event described in a previous machine. If there are no parameters, the keyword *any* is left out and the keyword *where* can be replaced with the keyword *when* for readability. A *witness* must be provided for each parameter which existed in an abstract event but has been replaced by a concrete implementation in the refined event. When all the *guards* evaluate to true the event is said to be *enabled*. This means that the event can occur and execute its *actions*, which are statements describing how the variables change.

Before we start describing how we create and refine our model using Event-B, we will look at the background of the application we will model. This includes the terminology used and the specifics of our approach.

```
event event-name ≙
  refines abstract-event
  any parameters
  where
    guards
  with
    witnesses
  then
    actions
end
```

Figure 2: *An event in Event-B. Based on Abrial [2].*

3 On-Demand Streaming

Streaming can be seen as the transport of data in a continuous flow, in which the data can be used before it has been received in its entirety. There are two different approaches to streaming content; live streaming and on-demand streaming. From an end user perspective live streaming is similar to a broadcast; that is, everyone who receives the media is intended to receive the same content at the same time. On-demand streaming is different, in that it is "essentially playback, as a stream, of prerecorded content" [21]. This makes on-demand streaming more similar to traditional file transfer. However, on-demand streaming is still "play-while-downloading" and not "open-after-downloading"[28], and traditional file sharing protocols can therefore not be used without modifications. This holds true especially if we look at peer-to-peer file sharing, where content is often transferred out-of-order. Although we will model media streaming in general, our intended target is the algorithms used in on-demand streaming solutions based on peer-to-peer file sharing protocols, and in particular the BitTorrent-based solution described by us previously [22]. We will therefore start by describing BitTorrent and how it can be modified for streaming.

3.1 BitTorrent and Piece Selection

BitTorrent is originally a peer-to-peer file sharing protocol and an application, designed by Bram Cohen and first released in July 2001 [8]. In the following years, BitTorrent evolved into one of the most popular peer-to-peer protocols [11, 14]. While the terminology used by Cohen [9] could be seen as a standard, the terminology used in this paper will be same as the one used by us previously [22] and based on that of Legout et al [16]. This terminology is close to that of the

application Vuze, formerly known as Azureus, a BitTorrent client often used as a basis for research applications [7, 12, 20].

When transferring data, BitTorrent needs to decide what piece of the content to request, i.e. piece selection, and which peers to send or not send data to, i.e. peer selection. The reference BitTorrent implementation starts by selecting pieces at random until one complete piece has been transferred [9]. The piece selection method is then switched to the rarest-first piece selection method, in which the piece with the lowest availability, i.e. the piece held by the fewest connected peers, is selected as the piece to request. This has the effect of reducing the likelihood that one piece may become unavailable, as peers will request the pieces with the lowest availability first. If more than one piece has the lowest availability, one of them is selected at random. When a single block from a piece has been downloaded, priority is given to the other blocks from that piece, in order to try to keep the number of incomplete pieces as small as possible. BitTorrent then continues requesting pieces using the rarest-first method until all pieces have been requested, at which time all remaining blocks are requested from all peers in the active peer set. This final step is done to prevent one slow peer from hindering the completion of the transfer.

To create an incentive for peers to upload as well as download, the reference BitTorrent implementation uses a tit-for-tat (TFT) mechanism for selecting peers. Based on the rate of data sent, the fastest peers are chosen for sending data to, and this is done every ten seconds. There is also one additional peer selected at random, re-evaluated every thirty seconds. This peer is called the "optimistic unchoke" [9, 16] and exists for two reasons: to allow newly joined peers to enter the TFT game, and to potentially discover faster peers that could become regular, non-optimistic unchokes [16, 20]. This approach is not necessarily the optimal way of peer selection. It has been shown that the TFT incentive mechanism can be exploited to allow peers to participate in a torrent without sending any data [17, 24]. Alternative approaches have been suggested, such as the one used by BitTyrant [20]. However, the basic TFT strategy remains an essential part of the BitTorrent protocol as used today.

After a peer has received all the pieces of the torrent, it may continue to participate in sending data to other peers, and in many cases the peer is actually encouraged to do so. When the peer thus has become a seed the peer selection method based on how much other peers have sent is of course not possible, and the reference BitTorrent implementation therefore switches to sending to the peers which can receive data the fastest [9]. However, as this feature could be exploited, later clients have switched to selecting peers randomly when seeding [20].

3.2 Piece Selection for Streaming

The original rarest-first piece selection method used in BitTorrent works well for file transfer, where the goal is the complete transfer and the order in which pieces

are received do not matter. However, when dealing with streaming media, out-of-order transfers are not optimal, as the content should be played back in-order. Therefore, the rarest-first piece selection method will not work as-is for streaming, and needs to be changed. Several different modifications to the BitTorrent protocol to enable streaming media have been proposed, for instance BiToS [26] and Give-to-Get [18]. We have chosen to model three different piece selection methods adapted for streaming.

The first piece selection method we chose to model was the distance-availability weighted method (DAW), described by us previously [22]. The idea behind DAW is to strike a balance between selecting lots of consecutive pieces and rarest pieces. The former is good for playback, and the latter is good for overall piece availability and for making the TFT mechanism work. In other words, we want to balance between requests between distance, i.e. the difference in piece number from the piece currently in playback to the given piece, and availability. While the distance and availability can be given different weights, we have kept them equal for the sake of simplicity. Pieces with low availability which are close to being played back are therefore selected before pieces which have high availability and are far from being scheduled for playback.

We begin by partitioning the pieces into those who are inside the buffer area, i.e. some k pieces after the one currently being played back, and those outside the buffer. The priority for requesting the pieces in the buffer will be the highest. Outside the buffer we will use the following formula for calculating the priority for each piece, ignoring pieces which have already been requested:

$$P = (p_s - p_c) * m_s$$

where m_s is the number of peers who hold a particular piece, p_s is the sequence number of that piece, and p_c is the sequence number of the currently playing piece. The piece which has the lowest P is selected, and if there are multiple pieces with the same value of P, we chose the one with the smallest distance.

It should be noted that when we talk about availability of a piece, we do not mean the amount of peers in total who are involved in the torrent and hold that particular piece. What we actually look at is how many peers our peer knows that hold that particular piece. As one peer need not necessarily be connected to all other peers, especially if the total number of peers is very large, we must by necessity look at the system from one peer's point of view.

Another way of doing piece selection for streaming is the rarest-first method with buffer (RFB), described by Vlavianos et al [26]. In essence, this is the rarest-first method from the original BitTorrent implementation, but with a fixed size buffer in which pieces are selected with the highest priority. The special cases for the first and last pieces in the original rarest-first method are also neither required nor implemented here, and if there is more than one piece with the same availability the one with the lowest piece number is selected first. This is the second piece selection we chose to model.

The third piece selection method we chose to model is the sequential method, representing a straightforward streaming solution. In this case, pieces are requested in the order they appear. While this may not work in conjunction with the TFT strategy for exchanging data, it is used when streaming content in the OneSwarm friend-to-friend sharing application [12].

4 Modelling with Event-B

While entire peer-to-peer systems and other distributed architectures have been formally modelled [13, 30, 29], we focus on modelling just the actual piece selection methods. The difference between these two approaches is that instead of looking at the whole network of peers, we model just how one peer looks at the system, with a focus on how the piece selection is done. Our idea when creating a model is to build it in separate layers, separating the functional parts from each other so that the model could easily be adopted for use with different functionality, e.g. different piece selection methods. Therefore, we intend to create all events as independent of each other as possible, leaving the concrete implementations of events to later refinements in order to prevent them from affecting other, less concrete events.

As we model our peer-to-peer client as a client for streaming media, we see that three major functions are needed; piece selection (possibly out-of-order), piece transfer (possibly out-of-order) and playback (always in-order). These three functions are independent of each other, but must be performed in this sequence. Hence, pieces must be selected before they are transferred, and pieces must be transferred before they can be played back. An example situation is hown in Figure 3. We require that selection will always happen at a greater rate than playback; that is, for each time we will advance playback we will have selected at least one additional piece.

4.1 Our Initial Model

We start from an abstract specification with events only for selecting pieces and advancing playback, and in these just counting how many pieces have been selected and played back. The constants and variables of the initial model, along with the corresponding axioms and invariants, can be seen in Figure 4. We note that we cannot play more pieces than we have selected (@*inv0_5*), and that we cannot complete until we have selected and played all pieces (@*inv0_6* and @*inv0_3*, respectively).

Our initial model contains five events: `INITIALISATION`, `SELECT`, `SELECT_AND_ADVANCE`, `ADVANCE` and `FINAL`. A representation of the event flow is shown in Figure 5, and the Event-B code representations of these events are shown in Figure 6. The three important ones are for selecting a piece, selecting a

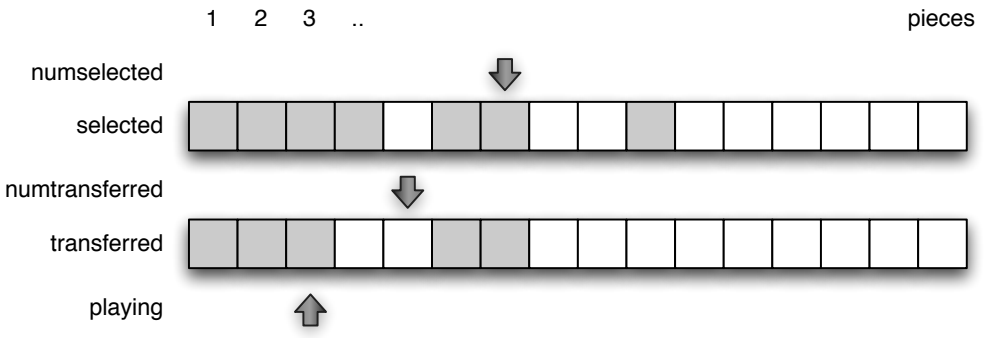

Figure 3: *The relation between* `selected`, `transferred` *and* `playing`. *The arrows indicate the number of pieces (7 selected, 5 transferred and 3 playing), while the gray squares indicate the specific pieces.*

context `PieceSelect_C0`
constants `pieces`
axioms
@*axm1* `pieces` $\in \mathbb{N}_1$
end

machine `PieceSelect_M0`
sees `PieceSelect_C0`
variables `playing completed numselected`
invariants
@*inv0_1* `playing` $\in$ `0..pieces`
@*inv0_2* `completed` $\in$ `BOOL`
@*inv0_3* `playing` $<$ `pieces` $\Rightarrow$ `completed` $=$ `FALSE`
@*inv0_4* `numselected` $\in$ `0..pieces`
@*inv0_5* `playing` $\leq$ `numselected`
@*inv0_6* `numselected` $<$ `pieces` $\Rightarrow$ `completed` $=$ `FALSE`
events
...
end

Figure 4: *The constants and variables of our initial model, and their corresponding axioms and invariants.*

piece and advancing playback, and just advancing playback. The model is done in this way because we require that selection will happen at a greater rate than playback, and therefore the action taken in each step can be that of selecting a piece, or selecting a piece and advancing playback. In other words, everytime something

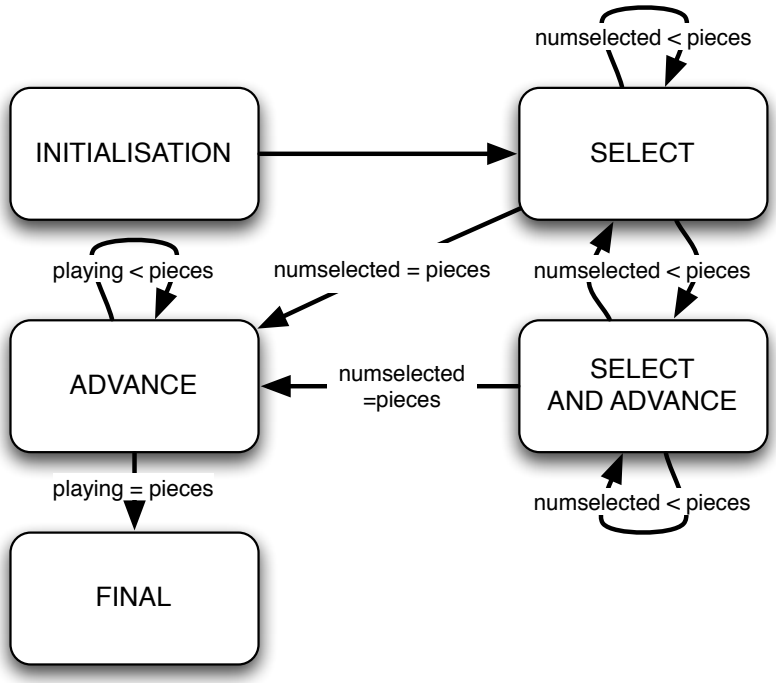

Figure 5: *A representation of the event flow of our initial model.*

happens in our model we will select a piece, and some of those times we also advance playback. The first event after `INITIALISATION` is always `SELECT`, because we cannot do a `SELECT_AND_ADVANCE` before we have received a piece to advance playback onto. The event `ADVANCE`, which only advances playback, is reserved for the case when all pieces have already been selected. The main purpose of the `FINAL` event is to show the requirements for successful termination. In this case that state is reached when all pieces have been both selected and played back.

4.2 The First Refinement

In the first refinement we introduce a new variable, `selected`, to keep track of exactly which pieces have been selected, in addition to the number of pieces as before. Because playback happens in-order, there is no need for a similar variable to keep track of which pieces have been played back. In Figure 7, the variables and invariants of the first refinement are shown. In addition to simple type constraints, we note that all pieces up to and including the currently playing one must have been selected (@*inv1_7*) and that if there is a piece which has not been selected, it implies that we have not selected all pieces (@*inv1_8*).

After this, we refine the `SELECT`, `SELECT_AND_ADVANCE` and `ADVANCE`

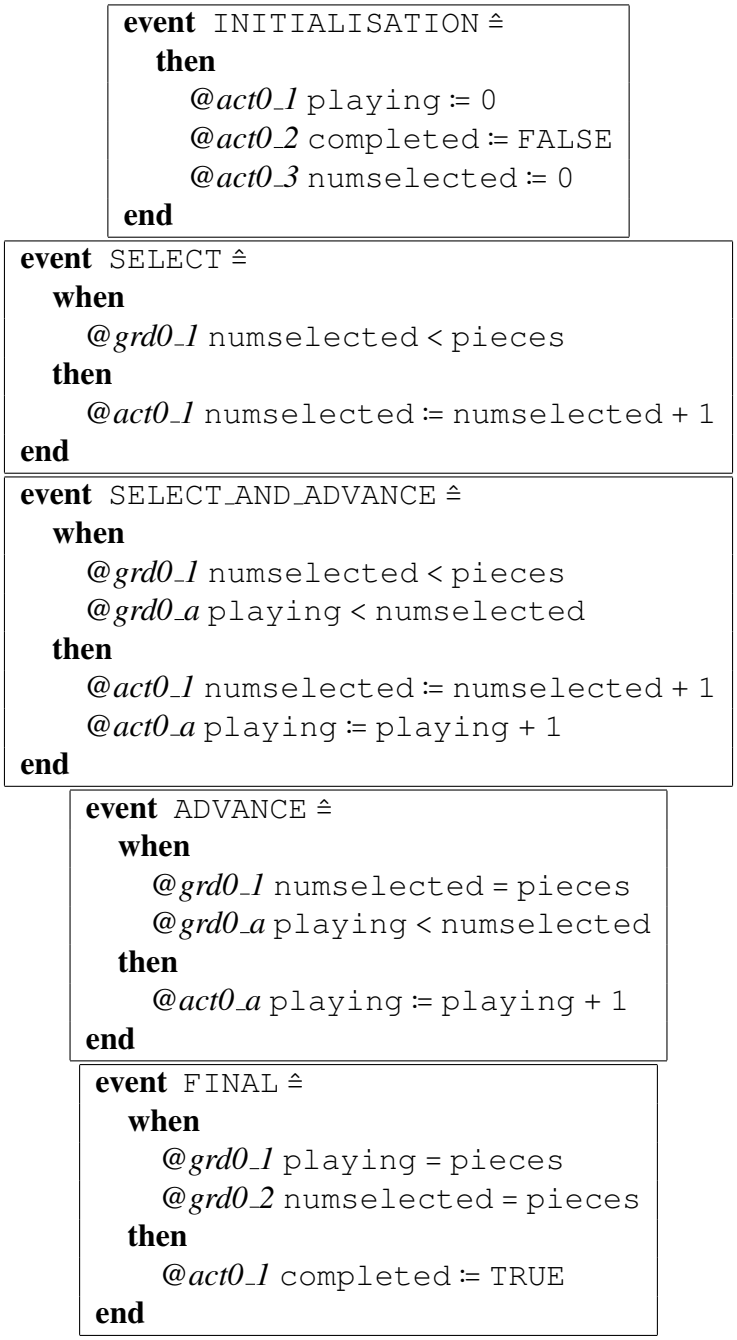

Figure 6: *The events of our initial model.*

events in the following way: the piece selected must not have been selected before (*@grd1_3*), and playback will only advance onto a piece if that specific piece has already been selected (*@grd1_b*). We also add the action which denotes a particular piece as selected (*@act1_2*). While the FINAL event is left as-is, the

machine PieceSelect_M1
  refines PieceSelect_M0
  sees PieceSelect_C0
  variables playing completed numselected selected
  invariants
    ...
    @*inv1_7* selected ∈ 1..pieces → BOOL
    @*inv1_8* ∀k · (k ∈ 1..pieces ∧ k ≤ playing
        ⇒ (selected(k) = TRUE))
    @*inv1_9* ∃s · (s ∈ 1..pieces ∧ (selected(s) = FALSE)
        ⇒ (numselected < pieces))
  events
    ...
end

Figure 7: *The variables of the first refinement of our model, and the invariants that were added in this refinement.*

INITIALISATION event is updated with setting selected to 1..pieces × FALSE. The refined events can be seen in Figure 8.

4.3 The Second Refinement

After the additions done in the previous refinement, we will add information about which pieces have been requested and which have been completely transferred. We will do this in two refinement steps, the first of which is this one. As seen in Figure 9, the context is extended to include a constant, simreq. This is the limit on how many outstanding requests we can have, i.e. how many pieces we can have selected but not transferred. To enable this we need a new variable, numtransferred, which keeps track of the number of pieces which have been completely transferred, and already in Figure 3 this function was described. Figure 10 shows the definition of numtransferred and the use of simreq as a limit on outstanding requests (@*inv2_12*). Furthermore, we require that the number of transferred pieces is less than or equal to the number of selected pieces (@*inv2_11*) and at the same time greater or equal than the number of pieces played back (@*inv2_13*).

The INITIALISATION event sets the number of transferred pieces to zero initially, and we introduce an event called TRANSFER for transferring a piece that has already been selected. These events can be seen in Figure 11. Furthermore, the events dealing with piece selection and advancing playback, i.e. SELECT, SELECT_AND_ADVANCE and ADVANCE, are refined by strengthening the guards.

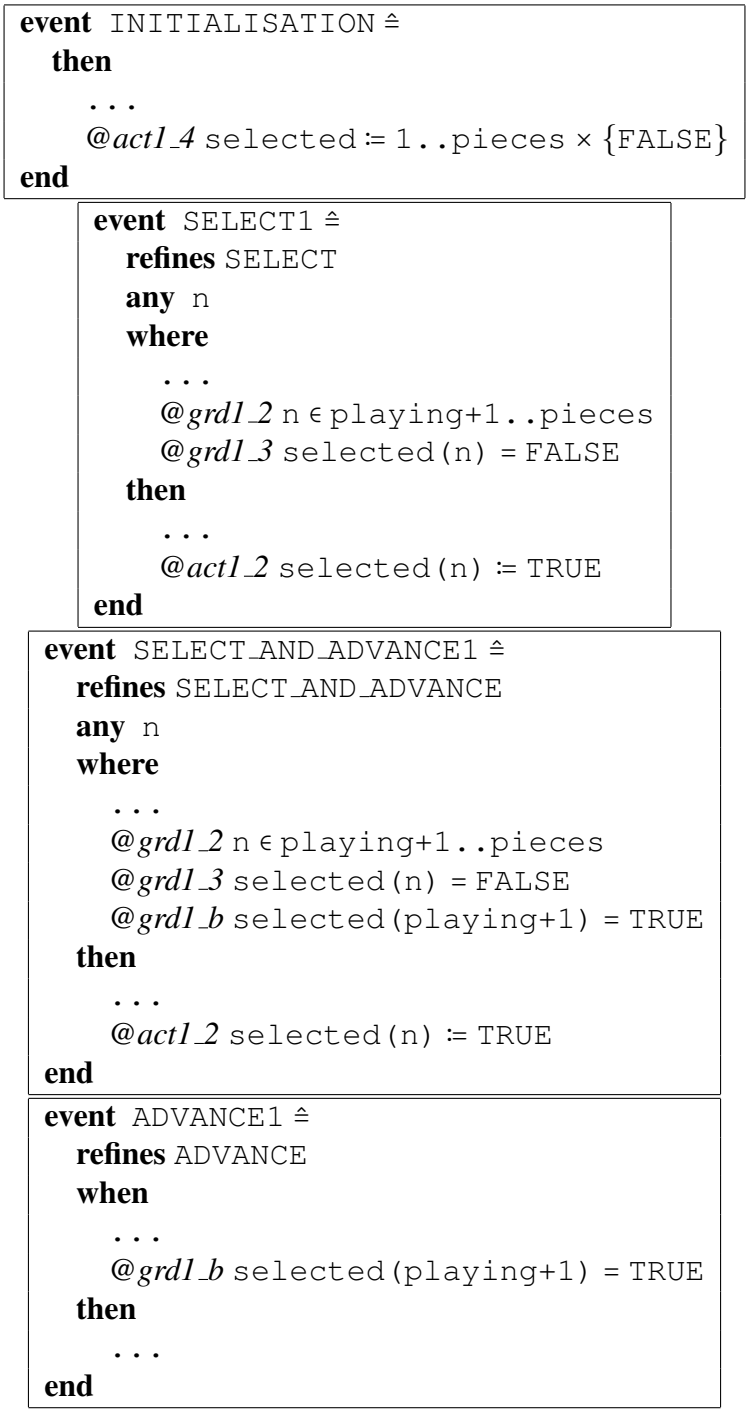

Figure 8: *The refined events in the first refinement of our model.*

The first two of these events gain a guard requiring that the difference between the number of pieces selected and the number of pieces transferred is less than

context PieceSelect_C2
  sees PieceSelect_C0
  constants simreq
  axioms
    @*axm1* simreq $\in \mathbb{N}_1$
end

Figure 9: *The constant and axiom that was added in the second refinement of our model.*

machine PieceSelect_M2
  refines PieceSelect_M1
  sees PieceSelect_C2
  variables ... numtransferred
  invariants
    ...
    @*inv2_10* numtransferred $\in$ 0..pieces
    @*inv2_11* numtransferred $\leq$ numselected
    @*inv2_12* numselected $-$ numtransferred $\leq$ simreq
    @*inv2_13* playing $\leq$ numtransferred
  events
    ...
end

Figure 10: *The variable and invariants that were added in the second refinement of our model.*

simreq (@*grd2_4*). The latter two of these events gain a guard requiring that the number of the currently playing piece is less than the total number of pieces transferred so far (@*grd2_c*). We also refine the FINAL event to only be enabled when all pieces have been transferred. These refined events can be seen in Figure 12.

4.4 The Third Refinement

In this refinement, we add to the the transfer functionality by introducing a new variable, numrequested, which is similar to the numtransferred variable introduced in the previous refinement in that it keeps track of the number of pieces requested. Additionally, two new variables, requested and transferred, are introduced to keep track of exactly which pieces have been

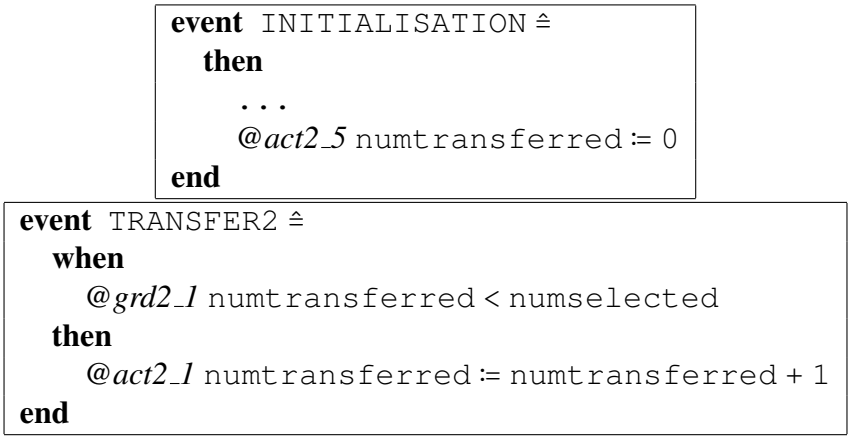

Figure 11: *The* INITIALISATION *and* TRANSFER *events in the second refinement of our model.*

requested and completely transferred, respectively. Thus, we now have variables to keep track of the number of pieces and the exact pieces for three steps taken on each piece, select (numselected and selected), request (numrequested and requested), and transfer (numtransferred and transferred). The fourth step, playback, only has a single variable, playing, associated with it, as playback always happens in-order. Figure 13 shows the variables of the third refinement of our model, and their corresponding invariants that have been added in this refinement step. We note that numrequested must always be between numselected and numtransferred (@*inv3_16* and @*inv3_17*). The context PieceSelect_C3 does not add anything to the previous one, and is therefore not shown.

The events in this third refinement of our model also gain these new variables. As seen in Figure 14, the INITIALISATION event sets the new variables to indicate that nothing has been requested or transferred initially. Because the addition of transferred, numrequested and requested does not affect piece selection, the SELECT event is not modified in this refinement. However, because we need to have transferred a piece before we can advance playback to that piece, the SELECT_AND_ADVANCE and ADVANCE events are refined to include a guard for this. Figure 15 shows these two events with the added guard (@*grd3_d*). Similarly, the FINAL event is updated with a guard requiring all pieces to have been requested before the event is enabled. This can be seen in Figure 16 (@*grd3_4*).

In the previous refinement, we introduced the TRANSFER event, and in this refinement we introduce a very similar event called REQUEST. These two events can be seen in Figure 17. The REQUEST event requests a piece that has been previously selected but not yet transferred (@*grd3_4* and @*grd3_5*), while the TRANSFER event is updated to require that before a piece can be transferred that

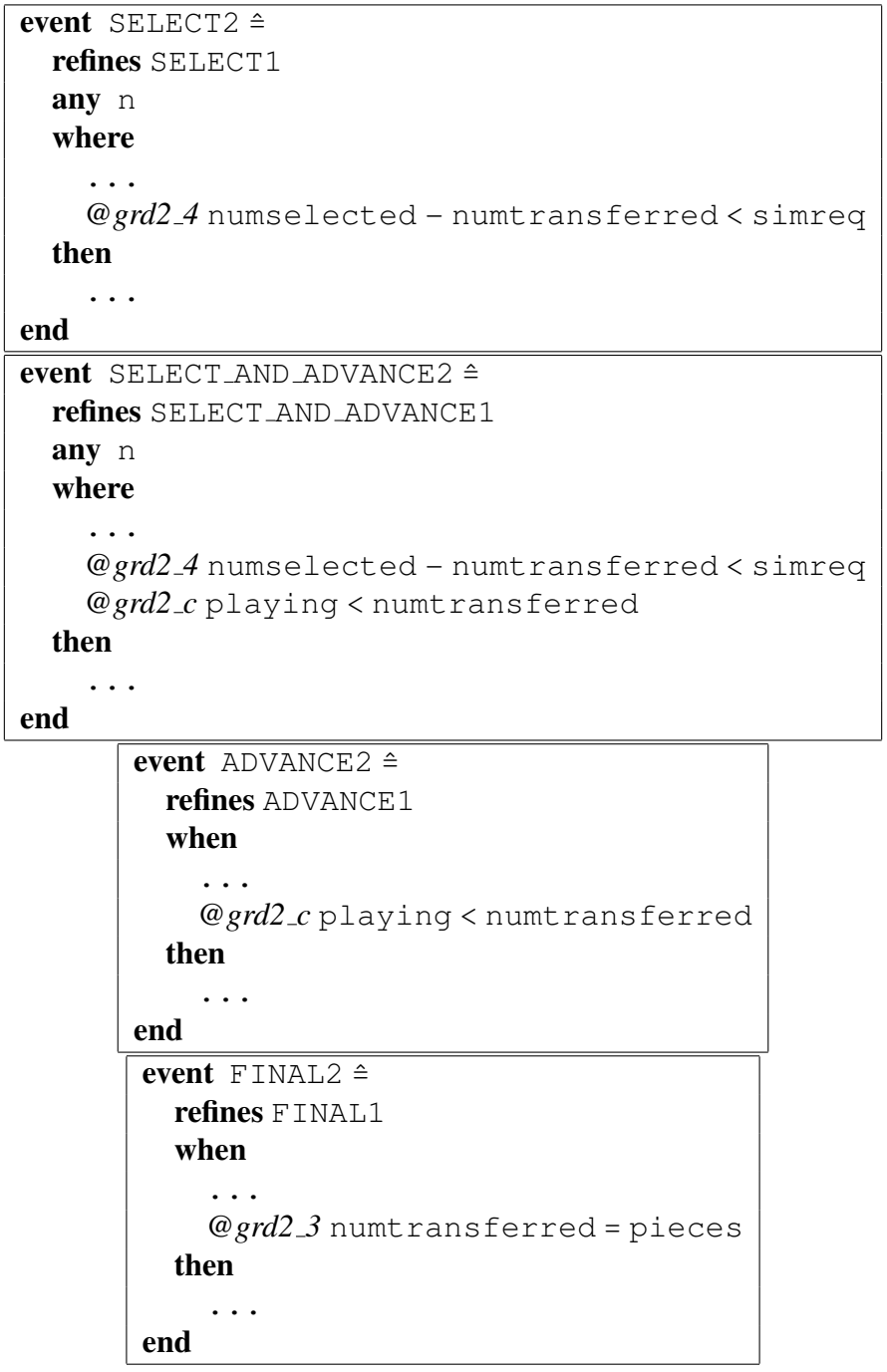

Figure 12: *The* SELECT, SELECT_AND_ADVANCE, ADVANCE *and* FINAL *events in the second refinement of our model.*

piece must be not only selected, but also requested, as well as not yet transferred (@*grd3_4*, @*grd3_5* and @*grd3_6*, respectively).

machine PieceSelect_M3
refines PieceSelect_M2
sees PieceSelect_C3
variables ... transferred numrequested requested
invariants
...
@*inv3_14* transferred ∈ 1..pieces → BOOL
@*inv3_15* numrequested ∈ 0..pieces
@*inv3_16* numrequested ≤ numselected
@*inv3_17* numrequested ≥ numtransferred
@*inv3_18* requested ∈ 1..pieces → BOOL
events
...
end

Figure 13: *The variables and invariants that were added in the third refinement of our model.*

event INITIALISATION ≙
then
...
@*act3_6* transferred ≔ 1..pieces × {FALSE}
@*act3_7* numrequested ≔ 0
@*act3_8* requested ≔ 1..pieces × {FALSE}
end

Figure 14: *The additions made to the* INITIALISATION *event in the third refinement of our model.*

So far, our refined machine is still very abstract, in the sense that it can be applied to any generic system which transfers content out-of-order and then uses the content in-order. In particular, the piece selection does not yet specify excatly how pieces whould be selected. In the following refinements we will work towards the concrete implementations of piece selection in BitTorrent streaming systems, starting with the Distance-Availability Weighted Method as described in Section 3.2. However, as piece selection methods generally use a concept of numeric priority, we will start by introducing priority in an abstract way in the next refinement, and requiring the piece selection to use it. After that we can model the priority in concrete ways as described by diffrerent piece selection methods. If we

event SELECT_AND_ADVANCE3 ≙
  refines SELECT_AND_ADVANCE2
  any n
  where
    ...
    @*grd3_d* transferred(playing+1) = TRUE
  then
    ...
end

event ADVANCE3 ≙
  refines ADVANCE2
  when
    ...
    @*grd3_d* transferred(playing+1) = TRUE
  then
    ...
end

Figure 15: *The* SELECT_AND_ADVANCE *and* ADVANCE *events in the third refinement of our model.*

event FINAL3 ≙
  refines FINAL2
  when
    ...
    @*grd3_4* numrequested = pieces
  then
    ...
end

Figure 16: *The* FINAL *event in the third refinement of our model.*

were to model a piece selection method which does not use numerical priority in the way we do, we would use the third refinement as our starting point.

4.5 The Fourth Refinement

In this refinement we introduce new variables for the purpose of describing priority. These variables, and their corresponding invariants, can be seen in Figure 18. The first one, priority, is a function from each piece number to a non-zero

event REQUEST3 ≙
  any j
  where
    @*grd3_1* numrequested < numselected
    @*grd3_2* numrequested < numtransferred + simreq
    @*grd3_3* j ∈ 1..pieces
    @*grd3_4* selected(j) = TRUE
    @*grd3_5* requested(j) = FALSE
  then
    @*act3_1* numrequested ≔ numrequested + 1
    @*act3_2* requested(j) ≔ TRUE
end

event TRANSFER3 ≙
  refines TRANSFER2
  any j
  where
    @*grd2_1* numtransferred < numselected
    @*grd3_2* numtransferred < numrequested
    @*grd3_3* j ∈ 1..pieces
    @*grd3_4* selected(j) = TRUE
    @*grd3_5* requested(j) = TRUE
    @*grd3_6* transferred(j) = FALSE
  then
    @*act2_1* numtransferred ≔ numtransferred + 1
    @*act3_2* transferred(j) ≔ TRUE
end

Figure 17: *The* REQUEST *and* TRANSFER *events in the third refinement of our model.*

natural number representing the priority for that piece (@*inv4_19*). The second one, priupd, contains the piece number of the last piece for which we updated the priority (@*inv4_20*). Because there is no need for updating the priority of pieces we have already played back (and therefore already selected, requested and transferred), we require that the number of the last piece for which we updated priorities to be at least as large as the currently playing one (@*inv4_21*).

In the INITIALISATION event, seen in Figure 19, the addition of new variables to the model requires new initialisations. The variable priupd is set to zero (@*act4_9*), because in the beginning we have not updated any priorities. The priority of each piece is set to one initially (@*act4_10*), but this value does not matter because we will update all priorities before they are actually used. Pri-

machine PieceSelect_M4
  refines PieceSelect_M3
  sees PieceSelect_C4
  variables ... priority priupd
  invariants
    ...
    @*inv4_19* priority $\in$ 1..pieces $\rightarrow \mathbb{N}_1$
    @*inv4_20* priupd $\in$ 0..pieces
    @*inv4_21* priupd $\geq$ playing
  events
    ...
end

Figure 18: *The variables and invariants that were added in the fourth refinement of our model.*

event INITIALISATION $\widehat{=}$
  then
    ...
    @*act4_9* priority $\coloneqq$ 1..pieces $\times \{1\}$
    @*act4_10* priupd $\coloneqq$ 0
end

Figure 19: *The additions made to the* INITIALISATION *event in the fourth refinement of our model.*

event CHANGE_PRIORITIES4 $\widehat{=}$
  any p
  where
    @*grd4_1* priupd < pieces
    @*grd4_2* p $\in \mathbb{N}_1$
  then
    @*act4_1* priority(priupd+1) $\coloneqq$ p
    @*act4_2* priupd $\coloneqq$ priupd + 1
end

Figure 20: *The* CHANGE_PRIORITIES *event in the fourth refinement of our model.*

orities will be updated by a new, abstract event called CHANGE_PRIORITIES, which can be seen in Figure 20. Initially, this event is enabled as long as we have not updated priorities for all possible pieces to request (@*grd4_1*) and where p is a non-zero natural number (@*grd4_2*). The priority of the piece after the previously updated one is then set to p (@*act4_1*), while the value of priupd is set to that piece (@*act4_2*). This means that we will update priorities of all pieces from the currently playing one to the last one.

The events REQUEST, TRANSFER and FINAL in this refinement are identical to the ones in the previous refinement, but the remaining events, SELECT, SELECT_AND_ADVANCE and ADVANCE, receive important additions, as can be seen in Figure 21. The first one is a guard requiring priorities to have been updated for all pieces before piece selection can happen (@*grd4_5*). Similarly, the value of priupd needs to be reset so that priorities can be updated again. In the SELECT event this means setting priupd to playing (@*act4_3*). However, in the SELECT_AND_ADVANCE event this means setting priupd to playing + 1 (@*act4_3*). The reason for this is that in this event the playback position is also advanced, as can be seen in Figure 6 (@*act0_a*). The ADVANCE event also gains an update to priupd, but here we set the value to pieces because we do not need to update priorities any longer (@*act4_b*).

The most important addition to the SELECT and SELECT_AND_ADVANCE events in the fourth refinement of our model is the addition of a guard regarding the priorities of pieces (@*grd4_6*). This requires the chosen n to be such that all possible pieces k which are not identical to n have a priority value equal to or higher than that of n. In practise, this means that the maximum priority that can be given to any piece is a numerical value of one, with higher numerical values being less prioritised. It also means that if there is more than one piece with the same priority, and that priority has the lowest numerical value of all priorities given to valid pieces, which one of these pieces to select is not determined. In our case, we will in the next refinement specify which of these pieces to select. However, as it is in this refinement the piece selection is actually consistent with the original BitTorrent specification, which does not specify an order when two or more pieces have the same availability [9].

4.6 The Fifth Refinement

So far our refinements have not been restricted to any particular piece selection method, although the last refinement introduced the concept of numerical priority and therefore restricted us to piece selection methods which can be modelled in such a way. Here, we will refine the model in different ways to focus on particular piece selection methods, starting with the distance-availability weighted method.

event SELECT4 ≙
  refines SELECT3
  any n
  where
    ...
    @*grd4_5* priupd = pieces
    @*grd4_6* ∀k · (k ∈ playing+1..pieces ∧ k ≠ n ∧
        selected(k) = FALSE ⇒ priority(n) ≤ priority(k))
  then
    ...
    @*act4_3* priupd ≔ playing
end

event SELECT_AND_ADVANCE4 ≙
  refines SELECT_AND_ADVANCE3
  any n
  where
    ...
    @*grd4_5* priupd = pieces
    @*grd4_6* ∀k · (k ∈ playing+1..pieces ∧ k ≠ n ∧
        selected(k) = FALSE ⇒ priority(n) ≤ priority(k))
  then
    ...
    @*act4_3* priupd ≔ playing + 1
end

event ADVANCE4 ≙
  refines ADVANCE3
  when
    ...
  then
    ...
    @*act4_b* priupd ≔ pieces
end

Figure 21: *The* SELECT*,* SELECT_AND_ADVANCE *and* ADVANCE *events in the fourth refinement of our model.*

4.6.1 The Distance-Availability Weighted Method

To model the distance-availability weighted method (DAW) from our already constructed abstract model, we need to refine the abstract priority introduced in the last refinement into a concrete one. As described in Section 3.2, the priority in

DAW is the highest in the buffer, which consists of a fixed number of pieces after the playing one. For other pieces the priority is calculated by multiplying the availability of the piece with its distance to the last piece in the buffer. Thus, we add a constant `buffersize` to describe the size of the buffer, and constants `minavail` and `maxavail` to describe the minimum and maximum values for piece availability. These constants can be seen in Figure 22. We assume that the availabilities must be larger than zero, because allowing zero availability for a piece would introduce additional complexity in piece selection and uncertainty whether all pieces could actually be transferred.

context PieceSelect_C5_daw
  sees PieceSelect_C4
  constants minavail maxavail buffersize
  axioms
    @*axm1* minavail $\in \mathbb{N}_1$
    @*axm2* maxavail $\in \mathbb{N}_1$
    @*axm3* minavail $\leq$ maxavail
    @*axm4* buffersize $\in$ 0..pieces
end

Figure 22: *The constants and axioms that were added in the fifth refinement when modelling DAW.*

We also need a new variable, `availability`, to describe the availability of each piece. This can be seen in Figure 23, along with the two invariants added. The first one of these simply states that the availability of each piece whould be between `minavail` and `maxavail` (@*inv5_22*). The second one states that when we have updated priorities for some but not all pieces, the pieces that we have updated priorities for and that are outside the buffer will have their priorities defined as distance times availability (@*inv5_23*).

Initially we set availability to `minavail` for all pieces, as seen in Figure 24. Because the availability is controlled by external forces, we need an abstract event which can change the availability of a piece. This event should be enabled independently of piece selection, but not be enabled when updating priorities, which depends on the availability of pieces. The new event `CHANGE_AVAILABILITY` can be seen in Figure 25. As described, the event is enabled for any valid piece (@*grd5_1*) and availability (@*grd5_2*) whenever we have updated priorities for all pieces (@*grd5_3*), and sets the availability of that piece (@*act5_1*).

The `CHANGE_PRIORITIES` event from the previous refinement is here refined into two separate events, `CHANGE_PRIORITIES5_BUF` for setting priorities for pieces in the buffer, and `CHANGE_PRIORITIES` for the other pieces. These events can be seen in Figure 26. The guard (@*grd5_3*) separates the two

```
machine PieceSelect_M5_daw
  refines PieceSelect_M4
  sees PieceSelect_C5_daw
  variables ... availability
  invariants
    ...
    @inv5_22 availability ∈ 1..pieces
        → minavail..maxavail
    @inv5_23 ∀t · (t ∈ playing+1..priupd ∧ priupd < pieces
        ∧ t > playing+buffersize
        ⇒ (priority(t) = (t − (playing+buffersize)) ∗
                                        availability(t)))
  events
    ...
end
```

Figure 23: *The variables and invariants that were added in the fifth refinement when modelling DAW.*

```
event INITIALISATION ≙
  then
    ...
    @act5_11 availability ≔ 1..pieces × {minavail}
end
```

Figure 24: *The additions made to the* INITIALISATION *event in the fifh refinement of our model when modelling DAW.*

different events, ensuring that only one of them is enabled at a time. In both events the parameter p from the abstract event is changed into a concrete one, necessitating the witness (@*p*) and removal of the guard stating the type of p (@*grd4_2* in Figure 20). In these events, p is replaced with the corresponding concrete priority according to the DAW piece selection method, as can be seen both in the witnesses and in the actions (@*act4_1*).

Of all the remaining events, the events ADVANCE, REQUEST, TRANSFER and FINAL do not need changing in this refinement. However, the SELECT and SELECT_AND_ADVANCE events both gain one guard. This guard corresponds to the requirement that if two pieces have the same priority, the one with the lowest piece number is selected. Figure 27 shows this guard in the

```
event CHANGE_AVAILABILITY5 ≙
  any n a
  where
    @grd5_1 n ∈ 1..pieces
    @grd5_2 a ∈ minavail..maxavail
    @grd5_3 priupd = pieces
  then
    @act5_1 availability(n) ≔ a
end
```

Figure 25: *The* CHANGE_AVAILABILITY *event in the fifth refinement when modelling DAW.*

SELECT_AND_ADVANCE event (*@grd5_7*), and because it is identical in the SELECT event that event is not shown.

4.6.2 The Rarest-First Method with Buffer

When modelling the rarest-first method with buffer (RFB), we can use our experience with modelling DAW as many parts are similar. In this fifth refinement of our model, the contexts of RFB and DAW are identical, and so are the variables and invariants of the machine. The CHANGE_AVAILABILITY event introduced in the fifth refinement for DAW is abstract enough that it can be used as-is for RFB as well, but the big difference is with the CHANGE_PRIORITIES event. Like in DAW, we refine the abstract event introduced in the fourth refinement of our model into two different events; one for pieces in the buffer and one for pieces outside the buffer. The CHANGE_PRIORITIES5_BUF event for pieces in the buffer is, again, identical to the DAW one, as they both assign the highest priority to buffersize pieces after the playing one. However, the CHANGE_PRIORITIES event for pieces outside the buffer is different. This event is shown in Figure 28. As before, the parameter p from the abstract event is replaced in this refinement with a witness stating the concrete value. In this case, the priority is set to availability of the piece in question (*@act4_1*), which results in pieces with low availability being prioritised over pieces with high availability.

The remaining events are either unchanged from the fourth refinement, as is the case with ADVANCE, REQUEST, TRANSFER and FINAL, or identical to the DAW ones. The requirement that if two or more pieces have identical priority the one with the lowest piece number must be selected is also present in RFB, and the corresponding SELECT and SELECT_AND_ADVANCE events are exactly as in Figure 27.

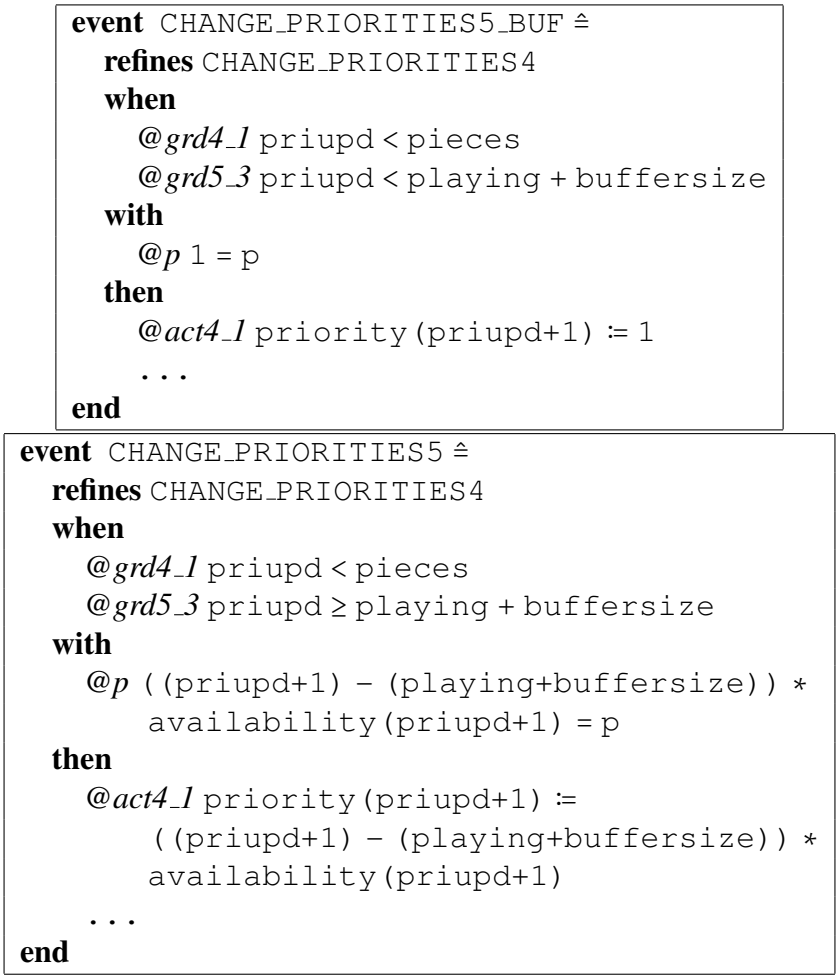

Figure 26: *The* CHANGE_PRIORITIES *events in the fifth refinement when modelling DAW.*

4.6.3 The Sequential Piece Selection Method

The sequential piece selection method is much simpler than the two previously described ones. Essentially, pieces are selected in order by setting the priority for a piece to its piece number. To model such a piece selection method we can use the fourth refinement of our model as a basis, without needing any new variables, constants or events. In fact, the only change is refining the CHANGE_PRIORITIES event. This can be seen in Figure 29.

As in the previous two final refinements, the parameter p from the abstract event is replaced by its concrete representation. In this case it is priupd+1, which is the piece number of the piece for which we are changing priority. This necessitates the addition of a witness (@*p*), and we also remove the type guard for p (@*grd4_2* in Figure 20).

```
event SELECT_AND_ADVANCE5 ≙
  refines SELECT_AND_ADVANCE4
  any n
  where
    ...
    @grd5_7 ∀j · (j ∈ playing+1..pieces ∧ j ≠ n
        ∧ selected(j) = FALSE
        ∧ (priority(n) = priority(j)) ⇒ (n < j))
  then
    ...
end
```

Figure 27: *The fifth refinement of the* SELECT_AND_ADVANCE *event when modelling DAW.*

```
event CHANGE_PRIORITIES5 ≙
  refines CHANGE_PRIORITIES4
  when
    @grd4_1 priupd < pieces
    @grd5_3 priupd ≥ playing + buffersize
  with
    @p availability(priupd+1) = p
  then
    @act4_1 priority(priupd+1) ≔ availability(priupd+1)
    ...
end
```

Figure 28: *The* CHANGE_PRIORITIES *event in the fifth refinement when modelling RFB.*

4.7 Proof Obligations

The Rodin platform automatically generates proof obligations, which define what needs to be proved for an Event-B model. The Rodin platform tool can automatically discharge most of these proof obligations by means of automatic provers, but some may need to be proved interactively, which the Rodin platform also provides the means for. Figure 30 shows the amount of proof obligations generated for each machine by version 2.0.1 of the Rodin platform tool, and how many of those that needed user interaction. The Rodin platform tool was used on a computer with a 2.4 GHz Intel Core 2 Duo processor running Mac OS X 10.5.8.

event CHANGE_PRIORITIES5 ≙
  refines CHANGE_PRIORITIES4
  when
    @*grd4_1* priupd < pieces
  with
    @*p* priupd+1 = p
  then
    @*act4_1* priority(priupd+1) ≔ priupd+1
    ...
end

Figure 29: *The* CHANGE_PRIORITIES *event in the fifth refinement of our model, when modelling sequential piece selection.*

Machine	Total Proofs	Interactive Proofs
PieceSelect_M0	18	0
PieceSelect_M1	34	0
PieceSelect_M2	44	0
PieceSelect_M3	64	0
PieceSelect_M4	71	2
PieceSelect_M5_daw	93	4
PieceSelect_M5_rfb	92	3
PieceSelect_M5_seq	72	2

Figure 30: *Proof Obligations of our Event-B Model.*

5 Conclusions

Using Event-B we have created a formal model of a node in a content transfer network, from the point of view of that node. We have refined our generic model into more specific ones, ultimately representing three different piece selection methods for media streaming. Our intent by creating a model using this method is to be able to reuse our generic model and adapt it in different ways. This gives us a foundation for creating formal models of different aspects of specialised distributed services. Previously we have been able to compare different distributed services using simulations [22]. However, having models refined as described here makes it possible to compare such services formally as well. Moreover, as consumers come to expect reliable and always available service infrastructure [6], we believe that our work is a step towards meeting these expectations. Future

work on the subject could include using our model as a basis for modelling other aspects of content transfer networks.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[4] R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools*, 13:26–39, 1996.

[5] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse. Online video using BitTorrent and HTML5 applied to Wikipedia. In *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'10)*, August 2010.

[6] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616, 2009.

[7] D. Choffnes and D. Bustamante. Taming the Torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In *Proceedings of ACM SIGCOMM 2008*, August 2008.

[8] B. Cohen. BitTorrent - a new P2P app. Yahoo eGroups, http://finance.groups.yahoo.com/group/decentralization/message/3160 (Accessed December 2010).

[9] B. Cohen. Incentives Build Robustness in BitTorrent. In *IPTPS*, 2003.

[10] Event-B and the Rodin Platform. http://www.event-b.org/ (Accessed December 2010).

[11] Ipoque Internet Study 2008 / 2009. Availabile from http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009 (Accessed December 2010).

[12] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Friend-to-friend data sharing with OneSwarm. Technical report, UW-CSE, February 2009.

[13] M. Kamali, L. Laibinis, L. Petre, and K. Sere. Reconstructing Coordination Links in Sensor-Actor Networks. Technical report, TUCS, February 2010.

[14] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. File-sharing in the Internet: A Characterization of P2P Traffic in the Backbone. Technical report, UC Riverside, November 2003.

[15] S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[16] A. Legout, G. Urvoy-Keller, and P. Michiard. Rarest First and Choke Algorithms Are Enough. In *Proceedings of ACM SIGCOMM/USENIX IMC2006*, October 2006.

[17] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *The 5th Workshop on Hot Topics in Networks: HotNets V*, pages 85–90, November 2006.

[18] J.J.D. Mol, J.A. Pouwelse, M. Meulpolder, D.H.J. Epema, and H.J. Sips. Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems. In *Multimedia Computing and Networking 2008, Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, Vol. 6818*, 2008.

[19] P2P Next. http://www.p2p-next.org/ (Accessed December 2010).

[20] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *4th USENIX Symposium on Networked Systems Design & Implementation*, 2007.

[21] P. Sandvik. Adapting Peer-to-Peer File Sharing Technology for On-Demand Media Streaming. Master's thesis, bo Akademi University, May 2008.

[22] P. Sandvik and M. Neovius. The Distance-Availability Weighted Piece Selection Method for BitTorrent: A BitTorrent Piece Selection Method for On-Demand Streaming. In *Proceedings of AP2PS '09*, October 2009.

[23] K. Sere. A Formalization of Superposition Refinement. In *Proceedings of the 2nd Israel Symposium on the Theory and Computing Systems*, June 1993.

[24] M. Sirivianos, J.H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent Networks with the Large View Exploit. In *Proceedings of IPTPS 2007*, February 2007.

[25] uTorrent Labs: Project Falcon. http://www.utorrent.com/labs/falcon (Accessed December 2010).

[26] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *9th IEEE Global Internet Symposium 2006*, April 2006.

[27] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design*, 13:5–35, 1998.

[28] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On Peer-to-Peer Media Streaming. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[29] L. Yan. A Formal Architectural Model for Peer-to-Peer System. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking 2010 Part 12*, pages 1295–1314. Springer US, 2010.

[30] L. Yan and J. Ni. Building a Formal Framework for Mobile Ad Hoc Computing. In *Proceedings of the International Conference on Computational Science (ICCS'04)*, June 2004.

List of Figures

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2535-2
ISSN 1239-1891