Formal Analysis and Verification of Peer-to-Peer Node Behaviour

Petter Sandvik[1,2] and Kaisa Sere[1]
[1]Department of Information Technologies, Åbo Akademi University
[2]Turku Centre for Computer Science (TUCS)
Joukahaisenkatu 3–5, 20520 Turku, Finland
{petter.sandvik,kaisa.sere}@abo.fi

***Abstract*—As services and applications move away from the one-to-many relationship of the client-server model towards many-to-many relations such as distributed 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. We have constructed a reusable formally derived and verified model of a node in a peer-to-peer network for on-demand media streaming, validated and animated it, and then compared the results with simulations. We have thereby created an approach for analysing peer-to-peer node behaviour.**

Keywords-formal modelling; peer-to-peer; BitTorrent; on-demand streaming.

I. Introduction

There has been a trend in computer software towards a "utility computing vision" [1] in which computer services are accessed without needing to know the specific underlying structure. Rather than the traditional client-server architecture of network services, this vision is largely dependent on many-to-many relations such as distributed cloud-based services and peer-to-peer systems. However, the recent increase in peer-to-peer usage has highlighted a few issues when it comes to development of such services. Testing a peer-to-peer system can be difficult and cumbersome, due to the often large scale and heterogeneous nature of the system. In some cases simulations can be used, but designing a thorough simulation is not an easy task. Furthermore, both testing and simulating these types of systems may require us to emulate the whole network of interacting nodes even if our interest would lie with only one of them, such as when developing a new application designed to interact with a network of existing ones.

Our background in formal methods made us wonder if this problem could be approached from the opposite direction. By this we mean that instead of testing and simulating the whole network to confirm the correct behaviour of one node, we create a formally verified model of one node and then use that model for analysing peer-to-peer node behaviour in general. We will here look at a peer-to-peer on-demand media streaming system, in which content is divided into pieces, distributed between peers using piece selection methods based on BitTorrent on-demand media streaming [2], and then played back in-order. We describe the creation of models for three specific piece selection methods, based on a common reusable formally derived and verified model in Event-B [3]. We then show how ProB [4] can be used to animate our Event-B model, giving results that we can compare with results from simulations. Hence, we show how these different techniques and tools complement each other in the design task.

The rest of this article is organised as follows: In Section II, we describe the Event-B formalism and the tools we have used, and in Section III, we give an overview of on-demand streaming. Section IV details the creation of our formal model. In Section V, we show the results of animation, and compare them with results from previous simulations. We conclude this article in Section VI with discussion and future work.

II. Event-B, the Rodin Platform and ProB

Event-B [5] is a formalism based on Action Systems [6], [7] and the B Method [8]. The primary concept in formal development with Event-B is *models* [5]. A model in Event-B consists of *contexts*, which describe the static parts such as constants and sets, and *machines*, which contain the dynamic parts such as variables, invariants (boolean predicates on the variables), and *events*. An event contains *actions*, which describe how the values of variables change in the event, and *guards*, which are boolean predicates that all must evaluate to `true` before the event can be enabled, i.e., able to execute.

In Event-B development starts from an abstract specification, and the model is then *refined* stepwise into a concrete implementation. In order to achieve a reliable system we use superposition refinement [9], [10] 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 [11]. In order to prove the correctness of each step of the development, we rely on the Rodin Platform [12] tool, which automatically generates *proof obligations*. These proof obligations, which are mathematical formulas that need to be proven in order to ensure correctness, can then be proven either automatically or interactively with the Rodin Platform tool. The choice of Event-B as the formalism to use for a model of this kind was largely due to this integrated tool support.

While the Rodin Platform tool is good for modelling and proving, we would also like to animate, or "execute" our models. This is because the mathematical correctness does not prove that our model does what we wanted it to do [5], and we would also like results that can be compared to those from simulations. ProB [4] is a free-to-use animator

 ISBN: 978-1-61208-173-1

and model checking tool, and supports models from both the B Method and Event-B. While ProB is available as a plug-in for the Rodin Platform tool, we have used the standalone, fully featured version for animation.

III. On-Demand Streaming

Streaming can be described 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 pre-recorded content" [13]. 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" [14], 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.

The basis for the peer-to-peer media streaming solution we will look at is the file sharing protocol BitTorrent [15]. In BitTorrent, content is partitioned into pieces of equal size, and by default these pieces are requested out-of-order. By modifying the algorithms used to select the order in which the pieces of the content is requested, i.e. piece selection, BitTorrent can be made to work for streaming content. Several different modifications to the BitTorrent protocol to enable streaming media have been proposed, for instance BiToS [16] and Give-to-Get [17]. We have here chosen to model three different piece selection methods; *sequential*, *rarest-first with buffer* (RFB) and *distance-availability weighted* (DAW). Sequential represents a straightforward streaming solution, requesting pieces in their original order. While this is used for instance when streaming content in the OneSwarm friend-to-friend sharing application [18], BitTorrent contains a tit-for-tat incentive mechanism that requires data to be out-of-order to function as intended, and the sequential method may therefore be of limited use when unknown peers are involved. RFB is a modification of the rarest-first method used in BitTorrent file sharing, where the piece held by the fewest other peers is requested. The addition of a buffer means that a specific number of pieces after the piece currently being played back are requested with the highest priority, thus striving towards always having a certain amount of the content immediately available for playback. and only after that will the rarest piece be requested. DAW [2] tries to strike a balance between requesting rare pieces and pieces that are close to being played back, by calculating priority using the distance (i.e., difference in sequence number between a specific piece and the currently playing one) multiplied with the availability. If there is more than one piece with the same priority, the piece with the lowest piece number, i.e. closest to being played back, will be chosen in both RFB and DAW.

For streaming to work, we note that data cannot be received slower than it should be played back, and this is something that we must take into consideration when creating our model. In the following section, we describe a common Event-B model for the piece selection methods, and refine that model into three specific ones corresponding to the three mentioned piece selection methods.

IV. Modelling with Event-B

Entire peer-to-peer systems and other distributed architectures have been formally modelled [19], [20], [21]. We have created a reusable Event-B model for a node in an on-demand content streaming network [3]. The difference between the two approaches is that instead of looking at the whole network of peers, we model just how one peer looks at the system. 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. Here we focus on modelling the piece selection methods.

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 shown in Fig. 1. As mentioned previously, the content must be transferred at least as fast as it should be played back in order to ensure that streaming works. Therefore, we require that selection will always take place at the same rate as playback or faster; that is, for each time we advance playback we will have selected at least one additional piece.

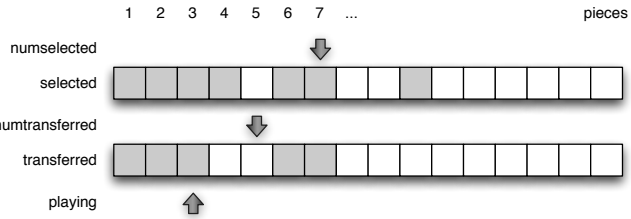

Fig. 1. *The relation between* `selected`, `transferred` *and* `playing`. *The arrows indicate the number of pieces (7 selected, 5 transferred and 3 playing), while the grey squares indicate the specific pieces.*

A. Common Model

We will start with a common model for all three piece selection methods [3], and here we will briefly describe the features of this model. We have two constants, `pieces` and `simreq`, which define how many pieces the content is divided into and the number of simultaneous requests, i.e., the maximum amount of pieces that can be selected but not yet transferred. We have variables for how many pieces we have selected (`numselected`) and exactly which pieces have been selected (`selected`), and similar variables for pieces that have been requested, i.e., transfer started, and for which the transfer has completed. We also keep track of which piece we are playing back and the priority for all pieces, as well as which piece we last updated priority for (`priupd`). The type restrictions of these variables are defined by invariants [3].

 ISBN: 978-1-61208-173-1

machine PieceSelect_M
variables playing completed numselected selected numtransferred transferred numrequested requested priority priupd
invariants
...
events
...
end

We initialise our common model with having zero pieces, and therefore not having selected, requested or transferred anything. Priorities are set initially for all pieces, but before the priorities are actually used they will be updated. This is done by an abstract event called CHANGE_PRIORITIES. which will be refined later. Initially, this event is enabled as long as we have not updated priorities for all possible pieces to request (@*grd4_1*) and for any p which 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.

event CHANGE_PRIORITIES ≙
any p
where
@*grd4_1* priupd < pieces
@*grd4_2* p ∈ ℕ1
then
@*act4_1* priority(priupd+1) ≔ p
@*act4_2* priupd ≔ priupd + 1
end

As the main focus in this model is piece selection, we will now look at the piece selection events. Because we require that content must be transferred faster than it should be played back, we also require that piece selection happens faster than playback. We have modelled this by separating the main flow of the program into two events: SELECT and SELECT_AND_ADVANCE. This means that the action taken in each step can be that of selecting a piece, or selecting a piece and advancing playback. In other words, every time something happens in our model we will select a piece, and some of those times we also advance playback. Naturally, after initialisation we always start with selecting a piece and only after that piece has been transferred could it be possible to advance playback.

The SELECT event is enabled when we have not yet selected as many pieces as we can (@*grd0_1* and @*grd2_4*) and when we have updated priorities for all pieces (@*grd4_5*). The parameter n must also be such that it can represent a piece we have not yet selected (@*grd1_2* and @*grd1_3*) and the priority for piece number n must be less than or equal to the priorities of all other possible pieces (@*grd4_6*). In practice, 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 refinements specify which of these pieces to select. However, the way the piece selection is modelled in this common model is actually consistent with the original BitTorrent specification, which does not specify an order when two or more pieces have the same availability [22].

What the SELECT event actually does is to increase the number of selected pieces (@*act0_1*), indicate that the specific piece has been selected (@*act1_2*) and reset the priupd variable so that we can update priorities before the next piece is selected (@*act4_3*).

event SELECT ≙
any n
where
@*grd0_1* numselected < pieces
@*grd1_2* n ∈ playing+1..pieces
@*grd1_3* selected(n) = FALSE
@*grd2_4* numselected − numtransferred < simreq
@*grd4_5* priupd = pieces
@*grd4_6* ∀k · (k ∈ playing+1..pieces ∧ k ≠ n ∧ selected(k) = FALSE ⇒ priority(n) ≤ priority(k))
then
@*act0_1* numselected ≔ numselected + 1
@*act1_2* selected(n) ≔ TRUE
@*act4_3* priupd ≔ playing
end

The SELECT_AND_ADVANCE event is very similar to the SELECT event, with the addition of guards and action concerning advancing playback. Thus, for this event to be enabled we require that we have not played all selected and transferred pieces (@*grd0_a* and @*grd2_c*), and that we have already selected and transferred the piece following the one currently being played back (@*grd1_b* and @*grd3_d*). The actions of this event are identical to the SELECT event, except for the addition of an action increasing the number of the currently playing piece (@*act0_a*) and therefore also requiring the increased value in the action that resets priority updates (@*act4_3*).

event SELECT_AND_ADVANCE ≙
any n
where
@*grd0_1* numselected < pieces
@*grd0_a* playing < numselected
@*grd1_2* n ∈ playing+1..pieces
@*grd1_3* selected(n) = FALSE
@*grd1_b* selected(playing+1) = TRUE
@*grd2_4* numselected − numtransferred < simreq
@*grd2_c* playing < numtransferred
@*grd3_d* transferred(playing+1) = TRUE
@*grd4_5* priupd = pieces
@*grd4_6* ∀k · (k ∈ playing+1..pieces ∧ k ≠ n ∧ selected(k) = FALSE ⇒ priority(n) ≤ priority(k))
then
@*act0_1* numselected ≔ numselected + 1
@*act0_a* playing ≔ playing + 1
@*act1_2* selected(n) ≔ TRUE
@*act4_3* priupd ≔ playing + 1
end

 ISBN: 978-1-61208-173-1

Our model also contains events which are not interesting in this context and therefore not shown here. We have events for pieces being requested and transferred, and in both we require that it is possible to perform the actions enabled by the event, which increase the number of requested or transferred pieces and mark the specific piece number as requested or transferred, respectively. The transfer event thereby updates variables that can be seen in the guards of the SELECT_AND_ADVANCE event. There is also an event that only advances playback after all pieces have been selected. We also have a final event which represents the conditions that must be true for the execution to terminate, which is that all pieces must have been selected, requested, transferred and played back. Only then will we set our variable completed to TRUE.

B. Three Piece Selection Models

Now that we have described our common model, we will take a look at our refined models which represent the use of three different piece selection methods.

1) The Sequential Piece Selection Method: The sequential piece selection method is very simple. 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 our common model as a basis, without needing any new variables, constants or events. In fact, the only change is refining the CHANGE_PRIORITIES event. The parameter p from the abstract event is here replaced by its concrete representation, 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.

event CHANGE_PRIORITIES_SEQUENTIAL ≙
  refines CHANGE_PRIORITIES
  when
    @*grd4_1* priupd < pieces
  with
    @*p* priupd+1 = p
  then
    @*act4_1* priority(priupd+1) ≔ priupd+1
    @*act4_2* priupd ≔ priupd + 1
end

2) The Rarest-First Method with Buffer: To model the rarest-first method with buffer (RFB) based on our common abstract model, we need to refine the abstract priority into a concrete one. As described in Section III, the priority in RFB is highest in the buffer, which consists of a fixed number of pieces after the playing one. Outside the buffer, the priority of each piece is set to the availability of that piece. 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. Availabilities must be larger than zero, because allowing zero availability for a piece would introduce additional complexity in piece selection and uncertainty as to whether all pieces could actually be transferred. We also need a new variable, availability, to describe the availability of each piece. We also add the following invariant, which 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 equal to their availability.

@*inv5_23* ∀t · (t ∈ playing+1..priupd ∧
  priupd < pieces ∧ t > playing+buffersize
  ⇒ (priority(t) = availability(t)))

Initially we set availability to minavail for all pieces. Because the availability is not controlled by us, we need an abstract event which changes the availability of a piece. This event should be enabled independently of piece selection, but not when updating priorities because they depend on the availability. The new event CHANGE_AVAILABILITY 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*).

event CHANGE_AVAILABILITY ≙
  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

For changing priorities, we refine our abstract event into two separate events, CHANGE_PRIORITIES_BUFFER for setting priorities for pieces in the buffer, and CHANGE_PRIORITIES_RFB for the other pieces. The guard (@*grd5_3*) separates the two different events, ensuring that only one of them is enabled at a time. 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. As can be seen both in the witnesses and in the actions (@*act4_1*), in these events the replacement for p is 1 and the availability of the piece, respectively.

event CHANGE_PRIORITIES_BUFFER ≙
  refines CHANGE_PRIORITIES
  when
    @*grd4_1* priupd < pieces
    @*grd5_3* priupd < playing + buffersize
  with
    @*p* 1 = p
  then
    @*act4_1* priority(priupd+1) ≔ 1
    @*act4_2* priupd ≔ priupd + 1
end

event CHANGE_PRIORITIES_RFB ≙
  refines CHANGE_PRIORITIES
  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)
    @*act4_2* priupd ≔ priupd + 1
end

 ISBN: 978-1-61208-173-1

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.

```
@grd5_7 ∀j · (j ∈ playing+1..pieces ∧ j ≠ n
    ∧ selected(j) = FALSE ∧
    (priority(n) = priority(j)) ⇒ (n < j))
```

The remaining events do not need refining.

3) The Distance-Availability Weighted Method: When modelling the distance-availability weighted piece selection method (DAW), we can use our experience with modelling RFB as many parts are similar. In this refinement, the contexts of RFB and DAW are identical, and so are the variables. However, the invariant concerning priority (*@inv5_23*) is different. Here, the priorities we have updated outside the buffer should be set to distance times availability [2].

```
@inv5_23 ∀t · (t ∈ playing+1..priupd ∧
    priupd < pieces ∧ t > playing+buffersize
    ⇒ (priority(t) = (t − (playing+buffersize))
    ∗ availability(t)))
```

The CHANGE_AVAILABILITY event introduced in the refinement for RFB is abstract enough that it can be used as-is for DAW as well, but the big difference lies in the CHANGE_PRIORITIES event. Like in RFB, we refine the abstract event from our common model into two different events; one for pieces in the buffer and one for pieces outside the buffer. The CHANGE_PRIORITIES_BUFFER event for pieces in the buffer is, again, identical to the RFB one, as they both assign the highest priority to buffersize pieces after the playing one. However, the event that changes priorities for pieces outside the buffer is different. The parameter p from the abstract event is here replaced with a witness stating the corresponding concrete priority according to the DAW piece selection method.

```
event CHANGE_PRIORITIES_DAW ≙
  refines CHANGE_PRIORITIES
  when
    @grd4_1 priupd < pieces
    @grd5_3 priupd ≥ playing + buffersize
  with
    @p ((priupd+1) − (playing+buffersize))
        ∗ availability(priupd+1) = p
  then
    @act4_1 priority(priupd+1) ≔
        ((priupd+1) − (playing+buffersize))
        ∗ availability(priupd+1)
    @act4_2 priupd ≔ priupd + 1
end
```

The remaining events are identical to the RFB ones, which in some cases means that they are unchanged from the common model. 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 DAW.

C. Proof Obligations

Event-B models have certain properties, and when refining a model these properties need to be preserved in order for the model to remain correct. The Rodin Platform tool generates proof obligations, which are the mathematical formulas that need to be proven in order to ensure this correctness, including preserving the invariants and strengthening of the guards of our Event-B models. The Rodin Platform tool can automatically discharge most of these proof obligations by means of automatic provers, but some may need to be proven interactively, which the Rodin Platform also provides the means for. Table I 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.

TABLE I
Proof Obligations of our Event-B Model.

Machine	Total Proofs	Interactive Proofs
PieceSelect_M	71	2
PieceSelect_M_SEQ	72	2
PieceSelect_M_RFB	92	3
PieceSelect_M_DAW	93	4

V. Validation, Animation and Comparison

As mentioned in Section II, we have used the standalone, fully featured version of ProB [4]. Due to the way our models are created and ProB interacts with them, memory and processing time constraints have forced us to use smaller values than we would in a real-life situation. Combined with limitations from simulations, this means that we look at the case when the content is divided into 20 pieces, only one request can be outstanding at any time, and the buffer size for RFB and DAW is set to 3. Although smaller than what would be used in a real-world situation, we believe that this is large enough to be noticeable but small enough not to impact the results.

We have compared the results from animating our models in ProB with the results from simple mathematical simulations [2], [13]. These simulations show the behaviour of the piece selection methods for the whole network, and here we chose a network of ten peers starting from scratch and one seed holding all the content. The results show how many of the peers hold each piece after twelve pieces have been selected, when selection happens twice as fast as playback.

Because our Event-B model looks at the network from the point of one node only, we have completed 40 random runs of the animation in ProB, and present the average results from these runs. The minimum availability was set to one and the maximum availability was set to five, and as in the simulation we have stopped to look at the situation after twelve pieces have been selected. Fig. 2 shows the results from both simulation and ProB animation for RFB and DAW piece selection methods.

The results for the sequential piece selection method are not shown, because the results from the simulation are identical

 ISBN: 978-1-61208-173-1

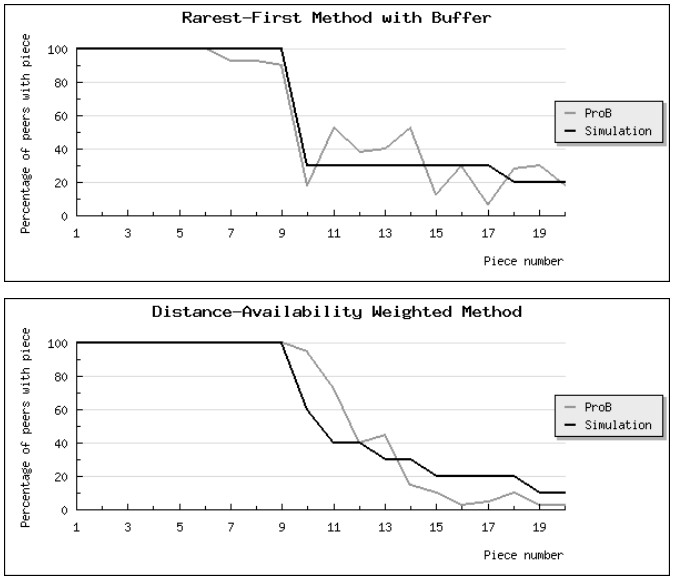

Fig. 2. *The availability of each piece after twelve pieces have been selected, when using RFB and DAW.*

to the ProB results, as should be expected. In both cases, the twelve first pieces are always selected after twelve pieces in total have been selected. For RFB and DAW, we note that the results from simulation and ProB animation are very similar. Some of the differences that exist are due to the fact that the simulations use the actual availability for each piece when calculating priority, while the animation uses random values corresponding to the nondeterminism in our Event-B model. Another difference regards the playback position. In the simulations, playback has always reached exactly piece number six after twelve pieces have been selected, because playback is advanced exactly every other time selection is done. In the animation, we start with selecting a piece, and after that we either select a piece or select a piece and advance playback with equal probability, which leads to variation in playback position but a mathematical average of 5.5. This means that unlike in the simulation, in the ProB animation an RFB node may not always have selected the 9 first pieces, and this is visible in Fig. 2. The very nature of DAW makes it unlikely, although not impossible, for the same thing to happen with DAW.

VI. Conclusions and Future Work

We have created a formally constructed and verified Event-B model of a node in a peer-to-peer content streaming network. This model we have then refined and animated, and the results have been compared with simulations. Using the same piece selection methods, our formal model of one peer has given us similar average results as a simulation of the whole network of peers. While we ran the ProB animation using smaller figures than would be used in a real-world situation, we still believe that our results show the added value that our method creates. By itself or together with simulations, animation of a formally derived and verified model can be used as an approach to analysing and verifying peer-to-peer node behaviour.

As our model is reusable, future work could include refining our model for other piece selection methods and extending the models and comparisons to other peer-to-peer systems besides an on-demand streaming one. Another possible direction would be refining our formal models to include the network structure in order to facilitate analysing aspects that require knowledge of more than just one node.

References

[1] R. Buyya, C. 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*, vol. 25, pp. 599–616, 2009.

[2] 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.

[3] P. Sandvik, K. Sere, and M. Waldén, "An Event-B Model for On-Demand Streaming," Turku Centre for Computer Science (TUCS), Tech. Rep. 994, December 2010, http://tucs.fi/publications/insight.php?id=tSaSeWa10a (Accessed September 2011).

[4] "The ProB Animator and Model Checker," http://www.stups.uni-duesseldorf.de/ProB/ (Accessed September 2011).

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

[6] R.-J. Back and K. Sere, "From Modular Systems to Action Systems," *Software - Concepts and Tools*, vol. 13, pp. 26–39, 1996.

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

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

[9] R.-J. 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*, 1983, pp. 131–142.

[10] S. Katz, "A Superimposition Control Construct for Distributed Systems," *ACM Transactions on Programming Languages and Systems*, vol. 15(2), pp. 337–356, April 1993.

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

[12] "Event-B and the Rodin Platform," http://www.event-b.org/ (Accessed September 2011).

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

[14] 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.

[15] B. Cohen, "BitTorrent - A New P2P App," Yahoo eGroups, http://finance.groups.yahoo.com/group/decentralization/message/3160 (Accessed September 2011).

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

[17] J. Mol, J. Pouwelse, M. Meulpolder, D. Epema, and H. 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.

[18] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson, "Privacy-Preserving P2P Data Sharing with OneSwarm," in *SIGCOMM'10*, August–September 2010, pp. 111–122.

[19] 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.

[20] L. Yan, "A Formal Architectural Model for Peer-to-Peer Systems," in *Handbook of Peer-to-Peer Networking 2010 Part 12*, X. Shen, H. Yu, J. Buford, and M. Akon, Eds. Springer US, 2010, pp. 1295–1314.

[21] M. Kamali, L. Laibinis, L. Petre, and K. Sere, "Self-Recovering Sensor-Actor Networks," in *FOCLASA*, 2010.

[22] B. Cohen, "Incentives Build Robustness in BitTorrent," in *1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

 ISBN: 978-1-61208-173-1