Petter Sandvik

A Formal Specification Language for Content Transfer Algorithms

Turku Centre *for* Computer Science

TUCS Technical Report
No 1109, April 2014

A Formal Specification Language for Content Transfer Algorithms

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

TUCS Technical Report
No 1109, April 2014

Abstract

When data is transferred between nodes in a network, it is often transferred in-order. However, in distributed systems, such as peer-to-peer networks and cloud-based systems, transferring data out-of-order can be advantageous, for instance by improving transfer speed, availability and reliability through the use of different algorithms. With the intent of creating a reusable formalism that can describe the complexities of out-of-order content transfer in a simple manner, while being powerful enough to support as large a variety of distributed content transfer algorithms as possible, we introduce the Specification for Content Transfer Algorithms (SPECTA) language. In this report, we also show how algorithms written in this language can be translated into other formalisms such as Event-B for analysis, verification or code generation.

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

In today's world, many applications use or require large amounts of data. In many cases, this data must be transferred from a network node to another. In systems using the traditional client-server model, this transfer usually happens in-order; that is, data is transferred in the same order it is stored. However, in distributed systems, such as peer-to networks and cloud-based systems, there are other ways of transferring data. Out-of-order content transfer algorithms may improve reliability, availability and transfer speed, and therefore there is a need to understand different content transfer algorithms.

Previously, content transfer algorithms have usually been described in a plain text format, easily understandable by humans, or in a formalism or program code that, although written by humans, is intended for machine interpretation. Examples of the former include BiToS [24] and DAW [19], and we have previously worked on the latter [21, 22]. In a more general case, describing what an algorithm does in a human readable format, especially for the purpose of learning, is often done using pseudocode, i.e., an informal high-level description of an algorithm resembling code in a programming language. Pseudocode has the unfortunate property of not being standardised, as well as, due to the way program flow is structured in different languages, being more or less tied to a specific programming language.

We aim to create a reusable formalism, in which we can describe the complexities of out-of-order content transfer in a manner that is easy to understand for anyone who is familiar with the problem solved by a specific algorithm, without requiring knowledge of what way the solution is implemented. At the same time, this formalism should be powerful enough to support as a large variety of distributed content transfer algorithms as possible. These can range from peer-to-peer video streaming to cloud-based file backup services, among others, but should also be able to support not yet foreseen situations in which describing these types of algorithms may be useful. Furthermore, we would like to be able to automatically translate from this formalism into other languages, not only for the purposes of modelling, verification, and generation of executable code, but also in order to be able to take advantage of the already existing tool support for these languages in the context they are used. For these purposes we introduce the Specification for Content Transfer Algorithms (SPECTA) language.

This report is organised as follows. In Section 2 we describe the SPECTA language. Section 3 contains a few examples of algorithms and their representation in SPECTA. In Section 4 we describe our ideas for translation from SPECTA to other languages, as well as an example of translation into Event-B. Section 5 concerns related work. We conclude this article in Section 6 with discussion about our results and possible future work.

2 The SPECTA Language

To be able to transfer content out of order, we need an initial ordering of the content, and a way of describing the order in which the content should be transferred. Therefore, we assume that the content to be transferred is partitioned into a finite number of pieces, enumerated sequentially by positive integers. Content transfer algorithms should then, based on properties of the pieces and of the system as a whole, step by step choose which piece to transfer next, until there are no eligible pieces left. Not all pieces are necessarily eligible at all times; for instance, while a backup solution may choose to backup the same content more than once in order to store content at different locations, a video streaming client usually does not need to transfer the same content more than once and therefore only needs to consider the pieces not already transferred. Deciding about eligibility of pieces is application-specific, and may even change over time, and in the algorithms we therefore do not specify which pieces are eligible.

When selecting the next piece to be transferred, we specify an overall condition that models a state the system must be in for the selection to take place at all, as well as criteria that describe the properties that an eligible piece must satisfy in order to be selected (1). The condition is separated from the criteria using a right triangle ($\triangleright$) symbol, and the criteria are separated from each other using a pipe ($|$) symbol. The use of a pipe symbol between criteria is due to its similar usage in UNIX-like command line environments, where the output of what is on the left-hand side of the pipe is used as the input for what is on the right-hand side. In SPECTA, the first criterion specifies a subset of all eligible pieces, the second criterion specifies a subset of the first subset, and so on. In the end, we are interested in no more than one piece at a time, and therefore, if the subset consists of more than one piece after the final criterion, we assume that one piece is non-deterministically chosen from the remaining subset.

$$\begin{aligned} \mathit{selection} \equiv \mathit{condition} \ \triangleright \ & \mathit{criterion}_1 \\ | \ & \mathit{criterion}_2 \\ | \ & \ldots \\ | \ & \mathit{criterion}_m, \text{ where } m \geq 1 \end{aligned} \tag{1}$$

If the condition is not fulfilled, or the condition is fulfilled but there is no eligible piece satisfying all criteria, no piece can be selected according to (1). Therefore, we need a way of combining several selection mechanisms, for which purpose we use the semicolon (2). If the first selection is possible, it is used, but if not, the second one is tried, and so forth. Unlike the selection

criteria in (1) there is no non-determinism involved if the final selection fails; we simply assume that the algorithm is designed in such a way that the correct behaviour is to do nothing until the conditions and criteria make it possible to perform a selection.

$$\begin{aligned} next \;=\; & selection_1; \\ & selection_2; \\ & \ldots; \\ & selection_k, \text{ where } k \geq 1 \end{aligned} \tag{2}$$

To express the conditions and criteria, we can use simple arithmetic operations, but we also need to define keywords and operations that can be used together with them. An important subset of these can be found in Table 1.

3 SPECTA Examples

With the keywords and operations of Table 1 in mind, we can give some examples. For instance, it is possible to write an in-order transfer algorithm by specifying that the piece with the lowest number should always be selected (3). Obviously, this requires that each previously requested piece is no longer eligible.

$$next = true \triangleright min(piece) \tag{3}$$

In the piece selection method used in the original BitTorrent peer-to-peer file sharing application [10], pieces are requested randomly until one complete piece has been transferred, and after that pieces are selected rarest-first, i.e., starting from the ones with lowest availability. The behaviour when there is more than one piece with the lowest availability is not specified [10], and therefore we leave the choice non-deterministic if the set of pieces with the lowest availability should contain more than one piece (4).

$$\begin{aligned} next = \; & transferred < 1 \triangleright random(pieces); \\ & transferred \geq 1 \triangleright min(avail(piece)) \end{aligned} \tag{4}$$

Main form	Alternative form(s)	Description
`piece`	`p`	The ID of the specific piece being considered.
`total`	`all, last`	The total number of pieces. Also the ID of the last piece, because pieces are numbered from 1.
`eligible`	`elig`	The set of all eligible pieces, a subset of all pieces.
`pieces`		The subset of eligible pieces satisfying all criteria so far. For the first criterion, this is the same as `eligible`.
`requested`	`r, req`	The number of pieces that have been requested.
`transferred`	`t, tr, transfered`	The number of pieces that have been transferred. This number may be smaller than `requested` when transferring content in parallel.
`availability(x)`	`av(x), avail(x)`	The number of nodes that hold the piece of content specified by the parameter. If no parameter, `piece` is assumed.
`minimum(x)`	`min(x)`	The piece (in `pieces`) for which the parameter is the smallest.
`maximum(x)`	`max(x)`	The piece (in `pieces`) for which the parameter is the largest.
`random(x)`	`random(x,y)`	If the parameter is a set of pieces, returns a random piece from that set. Otherwise gives a random number from `1` to `x`, or with two parameters, in the range from `x` to `y`.
`probability(r)`	`prob(r)`	Returns `true` with the probability `r`, $0 \leq r \leq 1$.
`size(x)`		Returns the size of the piece specified by the parameter.
`current`	`c, cur`	The current piece in any external use, e.g., playback position in media streaming.

Table 1: Keywords and operations

BiToS is a modification of BitTorrent to support streaming media [24]. This is done by partitioning the pieces into a high priority set, which consists of pieces close to being played back, and a set of pieces with lower priority. With a probability of 0.8 the rarest piece from the high priority set is chosen,

otherwise the rarest piece from the low priority set is chosen. In both cases, if there is more than one piece with the same availability, the piece with the lowest piece number is chosen. The optimal size of the high probability set was found to be 8% of the complete set of pieces [24]. With SPECTA we can describe the BiToS algorithm as (5).

$$\begin{aligned} next = prob(0.8) \ \triangleright \ & piece \leq current + (0.08 * total) \\ & \mid min(avail(piece)) \\ & \mid min(piece); \\ true \ \triangleright \ & piece > current + (0.08 * total) \\ & \mid min(avail(piece)) \\ & \mid min(piece) \end{aligned} \tag{5}$$

In our previous work, we have introduced another piece selection method for on-demand streaming media: the Distance-Availability Weighted method (DAW) [19, 21, 20]. In this method, pieces from the current media playback position up to a certain buffer size are requested sequentially. When the buffer is filled, pieces falling outside the buffer are prioritised based on their distance from the last piece in the buffer multiplied by their availability. Thus, pieces with low availability and close to being needed for playback are chosen before pieces that have high availability and are far from being played back. As in (5), if there is more than one piece that would receive the same priority, the piece with the lowest piece number is chosen. We can write the DAW algorithm in SPECTA as (6). Obviously, for this to be correct the eligible set would need to exclude any pieces with no availability, but in practice this would be the case anyway, as there is no point in requesting pieces that are unavailable.

$$\begin{aligned} next = true \ \triangleright \ & piece \leq current + buffersize \\ & \mid min(piece); \\ true \ \triangleright \ & min(avail(piece) \\ & \quad * (piece - (current + buffersize))) \\ & \mid min(piece) \end{aligned} \tag{6}$$

As another example, consider a distributed backup solution, in which each piece of content should be distributed to three different nodes. Moreover, all content should be available on n nodes before we start distributing it to $n+1$ nodes, and larger pieces of content should be transferred before smaller. This can be described as (7).

$$next = true \ \triangleright \ avail(piece) < 3 \\ \mid min(avail(piece)) \\ \mid max(size(piece)) \tag{7}$$

4 Translation from SPECTA

As previously mentioned, one of the ideas behind a language like SPECTA is to enable translation from a format easily understood by humans into a language meant for machine interpretation. The latter could be a programming language such as C or Java, depending on where the code is to be used. However, we have chosen to start by translating into Event-B [2], which is a formal modelling language. The reason for doing this is two-fold. Firstly, Event-B has excellent tool support in the form of the extensible Rodin Platform tool [13], which supports a variety of plug-ins for things such as animation and model checking [16] as well as code generation [17, 26]. Secondly, we have previously worked with content transfer algorithms in Event-B [21], thus facilitating easier comparison between automatically translated algorithms and those we have already modelled with Event-B. In the following, we will first describe Event-B and secondly present our approach to translation from SPECTA to Event-B and show examples of translated algorithms.

4.1 The Event-B Language

The B-Method [1] is a formal approach to specifying and developing highly dependable software, used successfully in development of several complex real-life applications [13, 11]. From the B-Method and the Action Systems [5, 7, 25] framework Event-B was derived for the purpose of modelling and reasoning about parallel, distributed and reactive systems, and the previously mentioned Rodin Platform [13, 3] was developed to offer tool support, which is an important asset for facilitating widespread adoption.

In Event-B, a model of a system is made up of two parts: a *machine*, usually referred to as the dynamic part of the model, and a *context*, seen as the static part of a model. Technically, an Event-B context is optional in a model, although in practice one is always included, as the context specifies constants, carrier sets, and axioms about these to be used in the model. An Event-B machine optionally *sees* a context, and holds the model state in *variables*, which are updated by *events*. An event is an atomic set of variable updates, happening simultaneously, and each event may also contain *guards*. The guards of an event are associated predicates that must evaluate to true

for the event to be able to execute, i.e., be *enabled*. If more than one event is enabled simultaneously, the choice between the events is non-deterministic. An Event-B machine should also include *invariants*, which are properties that must hold for any reachable state of the model. Thus, the properties specified by the invariants must hold before and after each occurrence of any event, after having been established by the **INITIALISATION** event. To be able to prove that this happens, the proof manager in the Rodin Platform [2, 13] tool automatically generates what needs to be proved in order for an invariant to hold.

Event-B provides a stepwise refinement-based approach to system development, where correctness is preserved by gradually introducing new variables, events and constants in a manner that does not disturb the previous functionality. Refinement starts from an abstract model, which describes what the system should do, and with each refinement step the system becomes more concrete, describing how it should do what it was designed to do. Refinement can be *horizontal* or *vertical*, where horizontal or superposition refinement [8, 15] refers to adding new variables, events and constants in addition to existing ones. Previous events can also be modified, typically such that they either update the newly introduced variables or introduce more deterministic assignments on the pre-existing variables, while also strengthening the event guards. Vertical refinement, or data refinement [6], corresponds to replacing some abstract variables with their concrete counterparts and accordingly changing the events. In this article, we do not use refinement, but it must be noted that our translation fits in somewhere between the most abstract model and the most concrete one. For instance, the algorithms written in SPECTA will give the number of the piece of content that should be requested for transfer, which can be seen as a refinement of an abstract event specifying that the "optimal" piece should be found. Likewise, in SPECTA we can specify the minimum or maximum of something within a set, which, conveniently, is a built-in function in Event-B, but a more concrete implementation would specify how the minimum or maximum should be found. In the following, we describe some of the other details that need to be considered when translating SPECTA into Event-B.

4.2 Translating SPECTA to Event-B

As both SPECTA and Event-B code are, to a certain extent, text-based formats, we decided to initially use a scripting language with good text processing abilities for translation. Therefore, we chose to write our prototype SPECTA to Event-B translator as a Perl script. We must stress the point that we do not translate SPECTA into Event-B models, nor into Event-B machines, but into Event-B code. Thus, the output of our translator cannot be used in the Rodin Platform tool without adding variables and other events

(in the machine) and relevant constants and axioms (in the context). This is done deliberately, as simply translating does nothing except show that it is possible to translate, and any practical use of the translated code in Event-B models should include additions of invariants and other events in such a way that there are real properties that should and could be proved.

When compared to SPECTA, Event-B has certain limitations. For instance, there is no support for random numbers or probabilities. In the former case we have opted to translate to the built-in non-determinism in Event-B, and for the latter we have translated such that we have a Boolean variable *probability* and an event that non-deterministically changes its value to true or false. This is because both randomness and probabilities can be seen as special cases of non-determinism; that is, anything we can prove about something that is true in a random way or with a certain probability we must be able to prove regardless of whether it is true or not. In other words, we now abstract probabilities and randomness into non-determinism, and assume that when or if Event-B can support these concepts natively we can reintroduce them as refinements of this non-determinism. Another problem we faced when translating is that Event-B currently only supports integer numbers, while SPECTA can work with real numbers, although we assume content pieces are numbered using integers. Together, these limitations mean that few algorithms written in SPECTA could be successfully translated completely without human intervention, although in most cases it will be possible to rewrite the algorithms in such a way that they can be translated, at least for the purpose of including them into a bigger model for the purpose of proving.

The first approach we tried for translating SPECTA to Event-B was a naïve implementation, in which each of the selections combined was translated into a single event. Initially, the first event, corresponding to the first selection, would be enabled, and the second event would be enabled if the first one selection failed, and so forth. This worked well for simple algorithms, such as ones consisting of only a few selections, each containing no more than two criteria. However, for larger algorithms this approach became unmanageable. The reason for this is as follows: in this implementation, each event had to specify all the conditions under which it would be enabled, and because the second event should only be enabled if the first one would not be able select a piece, the second event needed as its guard all the possible ways the first event could fail to select a piece. The third event would then only be enabled if the second failed to select a piece, resulting in even more combinations of conditions and criteria to include in the guards for the event. Thus, readability could become severely compromised and proving anything about the final model would also be more difficult. It was clear that we needed another approach.

The second approach we tried involved splitting the events from the first

approach into several ones. Instead of a single event for each selection, we had multiple events based on the different ways the previous selection could fail to select a piece. Thus, the first selection had one event, and the second selection had one event enabled when the first selection was unsuccessful because of the condition, and one additional event for each criteria that could cause the first selection to be unsuccessful. The third selection would then increase the number of events even further, based on the different combinations of how the the two previous selections failed to select a piece in order to enable the third selection. The large number of events did not only decrease the readability of the generated Event-B code, but would also increase the time needed to prove anything, as proving that an invariant holds needs to be done separately for each event. Therefore, it soon became obvious that this approach was not ideal either.

The third approach we tried, which is also the one we will describe here and show an example of, is based on the fact that the second selection onwards only need to know that the previous selections were unable to select a piece, not the reason why they were unable to do so. We could then replace the many events or complicated guards with a variable specifying which selection we were considering at the moment, knowing that if we considered selection method number two, selection method number one must for some reason have been unable to select a piece. This relationship could even be defined formally as an invariant stating that if we were considering the second selection, either the condition for the first selection must have been unfulfilled, or there was no eligible piece satisfying all criteria. In fact, this invariant must hold for all selections after the first one, and a similar invariant could be made regarding each selection method and made to hold for all subsequent ones.

A representation of program flow within the Event-B events created from SPECTA code can be seen in Figure 1. For each selection, we have two events representing whether the condition is fulfilled or not (for instance, **SP_SELECT_0** and **SP_SELECT_0_NEG**). To facilitate the use of multiple criteria in one selection, we also create a separate event for each criteria within each selection (**SP_SELECT_0_0**, **SP_SELECT_0_1**, ...). Each of these events defines *pieces* to be a new subset of the previous *pieces*, exactly as described in Section 2, but we also add one final event that, if *pieces* is non-empty, non-deterministically selects a piece from that set (**SP_SELECT_0_COMPLETE**). If the set *pieces* is empty at any time, meaning that one of the criteria is such that no eligible piece fulfills that one and all the previous ones, we reset the set *pieces* to contain all eligible pieces (in **SP_SELECT_EMPTY**), and move to the next selection. Likewise, if the final selection is not successful in selecting a piece, the event **SP_SELECT_FAILED** sets *pieces* to all eligible pieces and lets the program flow within the SPECTA-created events return to the idle state.

As mentioned previously, the events generated from SPECTA code do

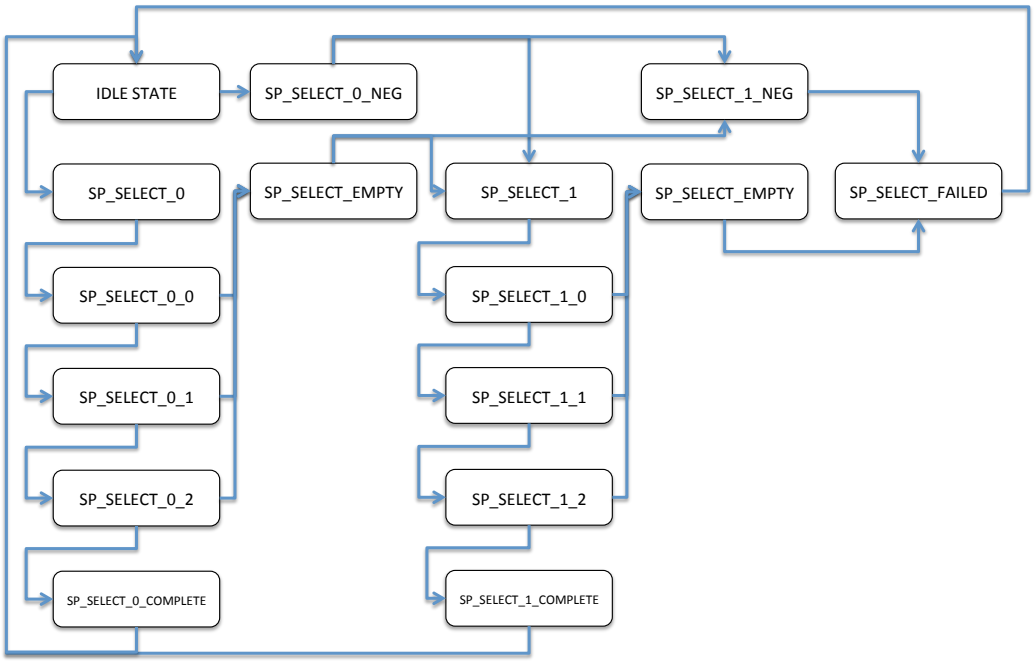

Figure 1: A representation of program flow within Event-B events created from SPECTA code, consisting of two selection methods each containing three criteria.

not constitute a complete model. For instance, we assume that the set of all pieces is constant, and therefore the number of pieces is also constant, and thus we can add these as constants in the context of our Event-B model. Likewise, *pieces*, *eligible*, *availability* and *next* are used by the events but not constant, and therefore we add them as variables to the Event-B machine and initialise them in the **INITIALISATION** event. We also have variables that model the program flow: *selection_method* describes which selection we are considering at the moment, roughly corresponding to the first digit or horizontal movement in Figure 1; *selection_step* describes which criterion we are looking at, roughly corresponding to the second digit or vertical movement in Figure 1; and *selection_inprogress* is a Boolean value representing whether we are doing piece selection, that is, whether we are outside the idle state in Figure 1 or not. The latter of these is done mainly to facilitate easier integration with events not generated from SPECTA code; if we need to prove anything about what happens in the piece selection events we need a way for other events updating variables such as *eligible* or *availability* to be prevented from doing so while piece selection is underway.

Because of space considerations, we will limit us to showing one example of SPECTA translated to Event-B events. The piece selection algorithm we show translated is DAW (6), previously presented in SPECTA form in Section 3. The first two events generated are shown in Figure 2. Already at

this stage there are two things to note. Firstly, Event-B differentiates between the logical true ($\top$) and the Boolean value true (TRUE), and thus we use the first in the guards translated from SPECTA but the second as a value that a variable can be set to. Secondly, in this example the condition is $\top$ and therefore the event requiring the negation as guard is superfluous since $\neg(\top)$ can never be true, although for the sake of completeness the event is still generated and we show it here.

event SP_SELECT_0
 where
  @grd1 $\top$
  @grd2 selection_method = 0
  @grd3 selection_step = 0
  @grd4 selection_inprogress = FALSE
 then
  @act1 selection_step := 1
  @act2 selection_inprogress := TRUE
end

event SP_SELECT_0_NEG
 where
  @grd1 $\neg(\top)$
  @grd2 selection_method = 0
  @grd3 selection_step = 0
  @grd4 selection_inprogress = FALSE
 then
  @act1 selection_method := 1
  @act2 selection_step := 0
  @act3 selection_inprogress := TRUE
end

Figure 2: Events corresponding to the condition of the first selection and its negation.

In Figure 3 we show the events corresponding to the criteria of the first selection, i.e., we first limit *pieces* to the pieces from *current* to *current* + *buffersize* (**SP_SELECT_0_0**), then we choose the piece from that set with the smallest piece number (**SP_SELECT_0_1**) and finally we choose one piece from the remaining subset (**SP_SELECT_0_COMPLETE**).

Figure 4 shows the event corresponding to the condition of the second selection, and as in Figure 2 we also have an unnecessary event which is only enabled when $\neg(\top)$ is true, i.e., never. However, we must point out that the reason this event can never be enabled here is because of the special situation that *true* is the condition for this selection method.

In Figure 5 we show the events generated based on the criteria of the second selection. As previously, the first guard of each event is the one that corresponds to the actual criteria, and here we note that **SP_SELECT_1_0** is quite complicated. Translation of a particular condition or criterion is done by using a separate file containing SPECTA statements used in conditions and criteria, and their equivalents in Event-B.

event SP_SELECT_0_0
 any newpieces
 where
  @grd1 newpieces = { piece | piece ∈ pieces ∧ piece < current + buffersize }
  @grd2 selection_method = 0
  @grd3 selection_step = 1
  @grd4 selection_inprogress = TRUE
  @grd5 pieces ≠ ∅
 then
  @act1 pieces := newpieces
  @act2 selection_step := 2
end

event SP_SELECT_0_1
 any newpieces
 where
  @grd1 newpieces = { piece | piece ∈ pieces ∧ piece = min(pieces) }
  @grd2 selection_method = 0
  @grd3 selection_step = 2
  @grd4 selection_inprogress = TRUE
  @grd5 pieces ≠ ∅
 then
  @act1 pieces := newpieces
  @act2 selection_step := 3
end

event SP_SELECT_0_COMPLETE
 where
  @grd1 selection_method = 0
  @grd2 selection_step = 3
  @grd3 selection_inprogress = TRUE
  @grd4 pieces ≠ ∅
 then
  @act1 next :∈ pieces
  @act2 selection_step := 0
  @act3 selection_method := 0
  @act4 selection_inprogress := FALSE
end

Figure 3: Events corresponding to the criteria of the first selection.

We note that even though the two events named **SP_SELECT_0_-COMPLETE** and **SP_SELECT_1_COMPLETE**, in Figures 3 and 5, respectively, appear to do the same thing, they could not be universally replaced by a single event. This is because in this particular case, both the first and second selection contain the same amount of criteria, which is not always the case. However, in Figure 6 one event that can be reached from any selection is shown (**SP_SELECT_EMPTY**). This event advances piece selection to the next method in the algorithm whenever criteria have made *pieces* an empty set. Figure 6 also shows the event that becomes enabled after the last piece selection method has failed (**SP_SELECT_FAILED**). In this event, *pieces* is set back to all eligible pieces and we indicate that piece selection is no longer in progress, thus creating a situation in which the variables that the piece selection methods depend on may be affected by

```
event SP_SELECT_1
  where
    @grd1 ⊤
    @grd2 selection_method = 1
    @grd3 selection_step = 0
    @grd4 selection_inprogress = TRUE
  then
    @act1 selection_step := 1
end

event SP_SELECT_1_NEG
  where
    @grd1 ¬(⊤)
    @grd2 selection_method = 1
    @grd3 selection_step = 0
    @grd4 selection_inprogress = TRUE
  then
    @act1 selection_method := 2
    @act2 selection_step := 0
end
```

Figure 4: Events corresponding to the condition of the second selection and its negation.

other events and thereby creating a situation in which next attempt at piece selection may be successful.

When putting this together with the mentioned variables, invariant specifying types of and relations between these, as well as an **INITIALISATION** event in the machine, and a context containing relevant constants and axioms about those, we created a complete and correct Event-B model that could be expanded upon, for instance by adding the properties we would want to prove and which would cause us to choose a formal modelling environment for development. We also acreated similar model for the BitTorrent piece selection algorithm (4), and all the proof obligations generated, for both the DAW and the BitTorrent models, were automatically discharged by the Rodin Platform tool.

4.3 Comparison

We have previously modelled a few different piece selection algorithms using Event-B [21, 22]. Therefore, we are able to compare the Event-B code that we created as a part of working with Event-B to the code now generated from SPECTA using our simple translator. One substantial difference is that in our previous work, we built a model of a distributed media streaming solution from the ground up, and there we introduced early on an event always selecting the most prioritised piece. In case of more than one piece with the same priority, the piece with the lowest piece number is selected, which is common to several piece selection algorithms for streaming media, such as (5) and (6), and has no effect in others, such as (3). Only in the final

```
event SP_SELECT_1_0
  any newpieces
  where
    @grd1 newpieces = { piece | piece ∈ pieces ∧
        (∀s · s ∈ pieces ∧ s ≠ piece ⇒
        (availability(s) ∗ (s − (current + buffersize)) ≥
        (availability(piece) ∗ (piece − (current + buffersize))))) }
    @grd2 selection_method = 1
    @grd3 selection_step = 1
    @grd4 selection_inprogress = TRUE
    @grd5 pieces ≠ ∅
  then
    @act1 pieces := newpieces
    @act2 selection_step := 2
end

event SP_SELECT_1_1
  any newpieces
  where
    @grd1 newpieces = { piece | piece ∈ pieces ∧ piece = min(pieces) }
    @grd2 selection_method = 1
    @grd3 selection_step = 2
    @grd4 selection_inprogress = TRUE
    @grd5 pieces ≠ ∅
  then
    @act1 pieces := newpieces
    @act2 selection_step := 3
end

event SP_SELECT_1_COMPLETE
  where
    @grd1 selection_method = 1
    @grd2 selection_step = 3
    @grd3 selection_inprogress = TRUE
    @grd4 pieces ≠ ∅
  then
    @act1 next :∈ pieces
    @act2 selection_step := 0
    @act3 selection_method := 0
    @act4 selection_inprogress := FALSE
end
```

Figure 5: Events corresponding to the criteria of the second selection.

refinement step we specified how the priorities would actually be calculated, such that we have separate events for different selection methods combined; for instance, in the case of DAW (6) we had one event setting the priority in the buffer and another event setting the priority outside the buffer. This way of working had the effect that when working with an in-order selection algorithm, each piece would always get the same priority in every iteration of the priority calculation, because in such a case priorities do not depend on any external information that could change.

In contrast to our previous work, the events generated by translation from SPECTA are not optimised for any particular case, and thus would be possible to use even with algorithms where calculating a numeric priority for each piece is not possible. Also, the way we translate from SPECTA makes

```
event SP_SELECT_EMPTY
  where
    @grd1 selection_inprogress = TRUE
    @grd2 pieces = ∅
  then
    @act1 pieces := eligible
    @act2 selection_step := 0
    @act3 selection_method := selection_method + 1
end

event SP_SELECT_FAILED
  where
    @grd1 selection_inprogress = TRUE
    @grd2 selection_method = 2
  then
    @act1 pieces := eligible
    @act2 selection_step := 0
    @act3 selection_method := 0
    @act4 selection_inprogress := FALSE
end
```

Figure 6: Events corresponding to some criteria not being able to select a piece and the whole algorithm not being able to select a piece.

it possible to use content transfer algorithms with large numbers of criteria for each selection method, and unlike our previous work we could also here utilise algorithms other than ones explicitly made for transfer of streaming media content.

5 Related Work

There has been a lot of work describing content transfer algorithms [10, 19, 20, 24], but not as much about the way these can be formally described. As mentioned in Section 1, pseudocode is not standardised, but there have been attempts at doing so [12]. There are also programming languages whose syntax is similar to pseudocode in that it is easily understood as a natural language, creating a situation where writing it requires learning a specific syntax but understanding existing code does not require specific knowledge about the syntax. One such language is AppleScript [4]. However, more relevant to the problem we have looked at is how to formally specify a domain-specific pseudocode [9]. More generally, we note that SPECTA is a domain-specific language, of which there are many [14]. Another related field is programming by construction [18], in which concepts can be specified in a manner not specific to any particular programming language, and then later used to generate code in specific programming languages.

6 Conclusion

In this paper we have introduced SPECTA, a language for specifying content transfer algorithms in an easily understandable manner. We have also shown how algorithms written this language can be translated into Event-B [2] as a starting point for analysis, verification, or further formal development. Based on our previous work, we have compared the formal model generated with converted SPECTA code to one developed separately, and noted the increased flexibility that comes with a generic automatic translation compared to that which created for a specific purpose.

Future work could include extending our language to support more information about other nodes, such as latency, transfer speed, and uptime. This would enable not only more advanced methods of selecting what piece of content to transfer, but also specifying which nodes should be involved. We also intend to rework our simple SPECTA to Event-B translator into a full-fledged plugin for the Rodin Platform [13] tool, in order to ease integration into the formal modelling workflow.

Acknowledgment

Initial parts of the work described in this article were presented as an extended abstract at the 24th Nordic Workshop on Programming Theory (NWPT 2012) [23].

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] J.-R. Abrial, M. Butler, S. Hallerstede, T.-S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[4] AppleScript Language Guide. `https://developer.apple.com/library/mac/documentation/AppleScript/conceptual/AppleScriptlangguide/AppleScriptLanguageGuide.pdf` (Accessed April 2014).

[5] 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.

[6] R.J.R. Back and K. Sere. Stepwise Refinement of Action Systems. *Structured Programming*, 12(1):17–30, 1991.

[7] R.J.R. Back and K. Sere. From Action Systems to Modular Systems. *Software - Concepts and Tools*, 17:26–39, 1996.

[8] R.J.R. Back and K. Sere. Superposition Refinement of Reactive Systems. *Formal Asp. Comput.*, 8(3):324–346, 1996.

[9] Michael Backes, Matthias Berg, and Dominique Unruh. A Formal Language for Cryptographic Pseudocode. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 353–376. Springer Berlin Heidelberg, 2008.

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

[11] D. Craigen, S. Gerhart, and T. Ralson. Case Study: Paris Metro Signaling System. In *Proceedings of IEEE Software*, pages 32–35. IEEE, 1994.

[12] John Dalbey. Pseudocode Standard. `http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html` (Accessed April 2014), 2003.

[13] Event-B and the Rodin Platform. `http://www.event-b.org/` (Accessed April 2014).

[14] Paul Hudak. Domain Specific Languages. In Peter H. Salas, editor, *Handbook of Programming Languages, Vol. III: Little Languages and Tools*, chapter 3, pages 39–60. MacMillan, Indianapolis, 1998.

[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] The ProB Animator and Model Checker. `http://www.stups.uni-duesseldorf.de/ProB/` (Accessed April 2014).

[17] V. Rivera and N. Catano. The EventB2Java Rodin Plug-in. `http://poporo.uma.pt/EventB2Java/EventB2Java.html` (Accessed April 2014).

[18] Mauno Rönkkö, Markus Stocker, Mats Neovius, Mikko Kolehmainen, and Luigia Petre. Programming by Construction. Technical Report 1092, TUCS – Turku Centre for Computer Science, 2013. `http://tucs.fi/publications/view/?id=tRxStNeKoPe13a` (Accessed April 2014).

[19] P. Sandvik and M. Neovius. The Distance-Availability Weighted Piece Selection Method for BitTorrent: A BitTorrent Piece Selection Method for On-Demand Streaming. In Antonio Liotta, Nick Antonopoulos, George Exarchakos, and Takahiro Hara, editors, *Proceedings of The First International Conference on Advances in P2P Systems (AP2PS 2009)*, pages 198–202. IEEE Computer Society, October 2009.

[20] P. Sandvik and M. Neovius. A Further Look at the Distance-Availability Weighted Piece Selection Method: A BitTorrent Piece Selection Method for On-Demand Media Streaming. *International Journal on Advances in Networks and Services*, 3(3 & 4):473–483, 2010.

[21] P. Sandvik and K. Sere. Formal Analysis and Verification of Peer-to-Peer Node Behaviour. In Antonio Liotta, Nikos Antonopoulos, Giuseppe Di Fatta, Takahiro Hara, and Quang Hieu Vu, editors, *The Third International Conference on Advances in P2P Systems (AP2PS 2011)*, pages 47–52. IARIA, November 2011.

[22] P. Sandvik, K. Sere, and M. Waldén. An Event-B Model for On-Demand Streaming. Technical Report 994, TUCS – Turku Centre for Computer Science, December 2010. `http://tucs.fi/publications/view/?id=tSaSeWa10a` (Accessed April 2014).

[23] Petter Sandvik. SPECTA: A Formal Specification Language for Content Transfer Algorithms. In Uwe Wolter and Yngve Lamo, editors, *24th Nordic Workshop on Programming Theory*, volume 403 of *Reports in Informatics*, pages 81–83. University of Bergen, 2012.

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

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

[26] S. Wright. Automatic Generation of C from Event-B. In *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3058-5
ISSN 1239-1891