



The United Nations
University

UNU/IIST

International Institute for
Software Technology

MultiScript I: The Basic Model of Multi-lingual Documents

Myatav Erdenechimeg, Richard Moore and
Yumbayar Namsrai

June 97

UNU/IIST and UNU/IIST Reports

UNU/IIST is a Research and Training Center of the United Nations University. It was founded in 1992, and is located in Macau. UNU/IIST is jointly funded by the Governor of Macau and the Governments of China and Portugal through contribution to the UNU Endowment Fund.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. advanced development projects in which software techniques supported by tools are applied,
2. research projects in which new techniques for software development are investigated,
3. curriculum development projects in which courses of software technology for universities in developing countries are developed,
4. courses which typically teach advanced software development techniques,
5. events in which conferences and workshops are organised or supported by UNU/IIST, and
6. dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research **[R]**, Technical **[T]**, Compendia **[C]** or Administrative **[A]**. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST or visit UNU/IIST home page: <http://www.iist.unu.edu>, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001



The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

MultiScript I: The Basic Model of Multi-lingual Documents

Myatav Erdenechimeg, Richard Moore and
Yumbayar Namsrai

Abstract

This document is the first of a series of three recording the output of the second phase of UNU/IIST's MultiScript project. It explains the basic structure of multi-directional, multi-lingual documents and some related auxiliary concepts (including how locations within such a document can be determined) and gives a formal specification of this in the RAISE specification language, RSL [6]. This modifies and extends the preliminary specification in [2, 3].

The other documents in this series cover the display and printing [5] and the creation and editing [4] of such documents.

Myatav Erdenechimeg worked as a UN Fellow at UNU/IIST from September 1995 until August 1996 and has continued at UNU/IIST as an Honorary Fellow since September 1996. She graduated from the National University of Mongolia with a degree in Mathematical Calculation and then became a lecturer in computer science at the same university. Her more recent work has concentrated on the problem of the computerisation of the Mongolian language. As part of this she has developed coding systems for both traditional Mongolian script and Mongolian Cyrillic letters and software for Mongolian word processing. She has also worked on software for translation between Mongolian script and Cyrillic. She has published several papers in these fields.

Richard Moore is a Research Fellow on the staff of UNU/IIST, a position he took up on October 1st 1995. He has an M.A. in mathematics from the University of Cambridge and a Ph.D. in physics from the University of Manchester. He has been engaged in computing science research in the field of formal methods since 1985, a large part of which was carried out in the formal methods group at Manchester University. He has written several papers on formal methods and is co-author of two books on formal methods – *mural: a Formal Development Support System*; and *Proof in VDM: A Practitioner's Guide*. He has also worked for the Defence Research Agency in Malvern, UK, on various formal methods projects, both as a consultant and as a full-time member of staff.

Yumbayar Namsrai is a UN Fellow at UNU/IIST from September 1996 to June 1997. He is the Head of the Programming Department in the Mathematics Faculty at The National University of Mongolia in Ulaanbaatar, Mongolia. He studied mathematics at the same university from 1967 to 1972, and worked in The Laboratories of Neutron Physics and of Computing Techniques and Automization at the Joint Institute for Nuclear Research (JINR), Dubna, USSR from 1972 to 1981 where he was awarded a Candidate of Science degree in Physics and Mathematics (Software Systems of Computers and Computing Systems) in 1982. His current research interests are in the computerization of the traditional Mongolian script.

Contents

1	Introduction	1
2	The Basic Model of Multi-lingual Documents	2
2.1	Embedded Text	2
2.2	Separated Text	4
2.3	The General Case	7
2.4	Documents and Libraries	8
3	The Formal Model of Multi-lingual Documents	8
3.1	The Basic Definitions	8
3.2	Consistency Conditions on a Document	11
3.3	Auxiliary Functions	12
4	Positioning within a Document	14
4.1	Auxiliary Functions	17
5	Some General Auxiliary Functions	18
6	Conclusion	19

1 Introduction

Broadly speaking, the majority of the world's scripts can be categorised into four main groups according to the direction in which they are read and written. We call these groups the "HL", "HR", "VL" and "VR" groups. Scripts in the HL group are written horizontally from left to right with lines of text proceeding downwards and include Latin, Greek, Cyrillic, Thai, and Bengali; those in the HR group are written horizontally from right to left, also with lines of text proceeding downwards, and include Arabic and Hebrew; those in the VL group are written vertically downwards in columns, the columns proceeding from left to right, and include traditional Mongolian script as well as related scripts like Manchu and Todo; and those in the VR group are written vertically downwards with the columns proceeding from right to left and include Japanese, Chinese and Korean.

However, despite this diversity in reading and writing direction, software systems which process multi-lingual documents typically tend to be unidirectional. Most commonly they impose the horizontal left-to-right directionality of the HL group, although some systems, particularly those which have primarily been produced to support a script which does not belong to this HL group, do adopt the natural directionality of that script and some indeed allow the user to change the reading and writing direction, though generally they neither allow this direction to change within a document nor support all four of the main reading and writing directions.

UNU/IIST's MultiScript project aims to design and prototype a multi-lingual document processing system in which the overall reading and writing direction of the document as a whole can be defined and where different parts of a document may also have different reading and writing directions. Moreover, the document as a whole or any part of it may have any of the four main reading and writing directions.

The first phase of the MultiScript project [2, 3] comprised a comprehensive study and analysis of multi-lingual documents, on the basis of which an abstract model of a generic multi-directional, multi-lingual document was developed. In addition, outline requirements for a software system supporting the creation, editing and display of such documents were formulated.

The second phase of the project extends and slightly modifies this basic abstract model of a document to include the notion of hyperlinks, and further generalises from a single document to a collection of libraries of documents. In addition, the model is extended to include specifications of how documents are displayed, printed, created and edited.

This document, the first of a series of three detailing the results of this second phase of the project, covers the basic model of multi-directional, multi-lingual documents, libraries and hyperlinks as well as some general auxiliary concepts. The other two documents in the series cover the display and printing [5] and the creation and editing [4] of such documents.

We begin by describing the principal components of the basic abstract model informally in Section 2, then we give a formal specification of this model in the RAISE specification language,

RSL [6], in Section 3. This model is extended in Section 4 to allow the position of individual elements within the contents of a document to be specified abstractly. Finally, we give specifications in Section 5 of some general auxiliary functions which are useful in both of the companion documents.

2 The Basic Model of Multi-lingual Documents

Our comprehensive survey of multi-lingual documents [2] identified two basic ways in which pieces of text in different scripts and possibly with different reading and writing directions can be intermixed in a single document: either pieces of text in one script are embedded within pieces of text in another so that the entire text is effectively read sequentially, or the pieces of text in the different scripts are clearly separated from each other and are read independently. We first deal with each of these cases separately, then go on to consider the more general situation in which both cases appear in the same document.

2.1 Embedded Text

When one script is embedded in another which has the same natural orientation this orientation is used by both. However, if the two scripts have different natural orientations then there are two different basic ways in which the insertion can be made: either the embedded script adopts the orientation of the script in which it is embedded or it retains its own natural orientation. In the former case the embedded script is often written with its characters rotated when one of the natural orientations is horizontal and the other vertical.

The first of these possibilities is illustrated by the first and third examples in Figure 1. The first example consists of English text in which Mongolian script is embedded. The Mongolian script does not have its natural orientation (vertical, top to bottom and left to right) but is instead written with the same orientation as the English text (horizontal, left to right and top to bottom), the individual Mongolian characters being rotated through 90° anticlockwise so that they are aligned correctly relative to each other when they are combined to form words (as in the last line of this example). The third example is similar to the first except that it shows Chinese text, written in its traditional orientation (vertical, top to bottom and right to left), in which English text is embedded. The English text is written with the same orientation as the Chinese, and again its individual characters are rotated, in this case through 90° clockwise, to ensure that the English words are correctly formed.

To model text with this kind of structure, we consider it as a sequence of pieces of *formatted text*, each of which has the same orientation (i.e. is written and read in the same direction) and consists of some *formatting information* and a sequence of *characters*. The formatting information effectively specifies the properties of each of the characters in that particular piece of text, for example their typeface, their size, their rotation, their colour, etc. In this way, the

To model text with this kind of structure, we consider each block of embedded text as being enclosed in a *frame*, with the reading and writing direction of the text within a frame being defined by the *orientation* of the frame. The orientation of a frame is in turn determined by its *entry point* and its *line stream*.

The entry point defines the position (for instance on a printed page of text or in a window on a computer screen) at which reading or writing within the frame begins, and is situated at the top left-hand corner of the frame for the HL and VL groups of scripts and at the top right-hand corner of the frame for the HR and VR groups. This definition could very easily be extended to include the bottom corners of the frame, thereby additionally supporting scripts which are read from bottom to top such as the Orkhon script (read horizontally, right to left and bottom to top; see [1], section 49), if so desired.

The line stream defines the direction in which the text within the frame is read or written, and is either horizontal (for the HL and HR groups of scripts) or vertical (for the VL and VR groups). Consecutive lines of text are aligned perpendicular to the line stream and proceed away from the entry point, so that for horizontal line stream they proceed downwards if the entry point is at the top and upwards if it is at the bottom while for vertical line stream they proceed to the right if the entry point is at the left and to the left if it is at the right.

The second example in Figure 1 is thus represented as a single piece of English text with three frames embedded in it, one containing each separate piece of Arabic text. Each of these three frames has right entry point and horizontal line stream and is read in its entirety when the appropriate point in the English text is reached. A frame embedded in a piece of text in this way is thus effectively considered as a single complex “character” within that text.

2.2 Separated Text

Text may appear in a document as separated blocks in two different ways. First, one block of text may logically follow some other text in the sense that it is intended to be read immediately after that other text. This is illustrated in the example in Figure 2 where a Japanese text, written vertically in columns and beginning at the top right-hand corner, is followed by its English translation.

In fact this type of example is just a special case of embedded text with different orientation as in the English/Arabic example discussed immediately above at the end of Section 2.1, the embedding simply coming at the end of the text. It is therefore modelled in exactly the same way: the English text is enclosed in a frame, whose entry point is left and whose line stream is horizontal, and that frame appears after the last character of the Japanese text (possibly with a line break character inserted between them).

The other way in which text may appear as separated blocks in a document is when the contents of one block have only a loose connection, or even no connection at all, with the contents or the

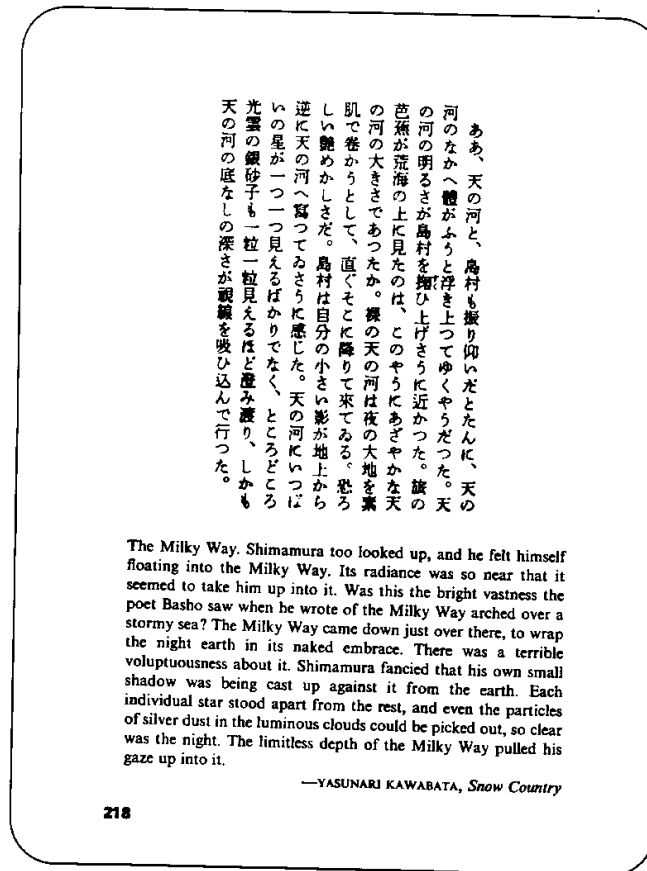


Figure 2: Japanese Text with English Translation

other. In this case, the two blocks, if loosely related, must appear “close” to each other but not necessarily at a given (fixed) position as would be the case for embedded text.

This is illustrated in Figure 3, which in fact shows a page from a German book about Mongolian grammar [7]. In this example, the text below the horizontal line, which consists of the German word ‘Leseübungen’ (which means ‘reading exercises’) together with a block of Mongolian script, is related to the text in §.201 above the horizontal line, but it could in principle appear anywhere within that section. In fact it is somewhat like a figure in the sense that its position may effectively float, at least within some bounds.

In order to model text with this kind of structure, we introduce the notion of a *free frame*. This has exactly the same properties as a frame used to describe embedded text apart from the fact that its position is not fixed absolutely in relation to the text with which it is associated when that text is printed or displayed.

2.3 The General Case

Of course in general a document may contain both embedded and separated pieces of text, as is in fact the case in the example shown in Figure 3, the part above the horizontal line consisting of German text with embedded Mongolian script (with its natural orientation: vertical, top to bottom and left to right), and the part below the horizontal line consisting of a separated piece of text loosely connected to the text in §.201 above the line. In general, therefore, we consider that a piece of text consists of two parts: a *string* and some *free frames*.

The string effectively represents the *ordered* part of the text, that is the part that is read sequentially from beginning to end, and consists of a sequence of *tokens*, each of which may be either a character or a frame. This sequence is split into pieces of formatted text as described in Section 2.1 in order to accommodate changes in typeface, character rotation, colour, etc. within the text. Frames within the sequence contain embedded text whose reading and writing direction is different from that of the containing text.

Conversely, the free frames represent the *unordered* part of the text, that is things which have only a loose connection, or even no connection at all, with the ordered part of the text or with each other. A free frame might contain text, as in the example shown in Figure 3 and discussed in Section 2.2, but more generally it might also, for example, contain a figure, a picture or a table. A good example of a document which only contains free frames is a newspaper. This essentially consists of a set of unrelated or loosely related individual articles, possibly with photographs or other pictorial items, so it can be represented simply as a collection of free frames, one for each individual article or pictorial item.

When a piece of text is printed or displayed, the position of each object in the string part is completely determined by the orientation of the text and by the position and size of the preceding object in the string, if any, and frames embedded in the string act just like characters, albeit probably ones of abnormal size. The free frames, on the other hand, can effectively be positioned at will.

The whole of the example shown in Figure 3 is thus represented as an ordered string part plus one free frame. The string part represents the text above the horizontal line, that is the German text with the embedded Mongolian, and the free frame represents the part below the horizontal line as explained in Section 2.2. The changes in typeface between normal and italic characters in the German text are effected by dividing the string into a sequence of pieces of formatted text, and each individual piece of Mongolian text is represented as an embedded frame within this string. The text within the free frame consists only of a string part, this in turn consisting of a single piece of formatted text comprising the characters of the German word 'Leseübungen' followed by a frame containing the block of Mongolian text. The Mongolian text also consists only of a string part constructed from a single piece of formatted text, this formatted text being made up from the individual Mongolian characters in sequence.

2.4 Documents and Libraries

As we have seen above in the German/Mongolian example of Figure 3, a frame may contain other frames or *subframes*. In fact these may occur in the string part or in the free frames part, and could in principle be arbitrarily nested with subframes also containing subframes. We can therefore make use of this nesting and simply consider a whole document as a frame, the orientation of that frame determining the sense in which the document should be read and the contents (string and free frames) of the frame representing the text within the document.

In this way, the German/Mongolian document [7] as a whole, of which Figure 3 shows one page, is represented as a frame with left entry point and horizontal line stream (corresponding to the natural orientation of the German text, i.e. of the main text of the document), the contents of that frame being a string containing the main German text with the embedded Mongolian (as in the upper part of Figure 3) and a collection of free frames representing the various exercises associated with the different subsections of the main text (as in the lower part of Figure 3).

Documents are collected together in *libraries*, each library consisting of a set of documents, and the ultimate scope of our model is a collection of libraries. Each library might, for example, represent a collection of documents owned by a particular individual, with the collection of libraries as a whole representing all documents owned by all individuals.

3 The Formal Model of Multi-lingual Documents

We now develop a formal specification of multi-directional multi-lingual documents based on the informal analysis given in Section 2. This formal specification is given in the RAISE specification language RSL [6].

3.1 The Basic Definitions

We define a type ‘Doc’ as an RSL *record* type containing three components, one for each of the three basic constituents of a document: its *root*, its *frame map*, and its *hyperlinks*. These fields must satisfy various consistency conditions in order for the type to represent only valid (or well-formed) documents, so we define the type ‘Document’, which represents well-formed documents, as a *subtype* of the type ‘Doc’, specifically as those values of type ‘Doc’ which satisfy the predicate ‘is_wf_document’. The discussion of these consistency conditions on a document and the definition of the function ‘is_wf_document’ are given in Section 3.2.

The frame map of a document effectively records the properties and contents of each frame (both embedded subframes and free frames) in the document. This is modelled as a *map* type in RSL, from an abstract type (or *sort*) ‘FrameId’ of *frame identifiers* to the type ‘Frame’ which

represents the actual frame, to make it possible to distinguish different frames which happen to have the same orientation and contents.

The root of a document represents the frame identifier of the topmost (or main) frame of that document, that is of the frame which corresponds to the document as a whole.

Finally, the hyperlinks field records referential links from some part of a document to some other part of the same document or to some part of some other document, possibly in some other library. This is modelled as a map type from frame identifiers in the current document to other frame identifiers, the *library identifier* (represented by the abstract type ‘LibId’) and the *document identifier* (represented by the abstract type ‘DocId’) components of the *product* type in the range of the map indicating the library and the document within that library to which these latter frames belong.

```

type
  Doc ::
    root : FrameId
    frame_map : FrameId  $\overrightarrow{m}$  Frame
    hyper_links : FrameId  $\overrightarrow{m}$  (LibId  $\times$  DocId  $\times$  FrameId),
  Document = { | d : Doc • is_wf_document(d) | },
  FrameId,
  DocId,
  LibId

```

The document identifier distinguishes individual documents in a library, and a library is then simply represented using a map type from document identifiers to documents. Similarly, the library identifier distinguishes individual libraries within the whole collection of libraries, represented by the type ‘Libraries’.

```

type
  Library = DocId  $\overrightarrow{m}$  Document,
  Libraries = LibId  $\overrightarrow{m}$  Library

```

The type ‘Frame’, which represents frames, is also modelled as a record type. This consists of three parts: the orientation of the frame, represented by the type ‘Orientation’; its string, represented as a list of pieces of formatted text, each piece represented by the type ‘FormattedText’; and its free frames, represented as a list of frame identifiers. The string and the free frames together constitute the contents of the frame.

```

type
  Frame ::
    orient : Orientation
    string : FormattedText*
    freeFrames : FrameId*

```

The orientation is defined as a record type comprising an entry point and a line stream. Both entry point and line stream are defined using *variant* types which simply enumerate all possible values of those types. As pointed out in Section 2.1, this definition of the entry point could easily be extended to include scripts like the Orkhon script which are written from bottom to top, specifically by extending the variant type definition to additionally include the values ‘bottom_left’ and ‘bottom_right’ (for example).

type

```
Orientation :: entry : Entry_Point stream : Line_Stream,
Entry_Point == left | right,
Line_Stream == horizontal | vertical
```

Formatted text is also defined as a record type consisting of some formatting information and a list of tokens. A token is either a character or a frame identifier, as represented by the *union* type. Formatting information and characters are defined only abstractly as sorts.

```
FormattedText :: format : FormatInfo chars : Token*,
Token = Character | FrameId,
FormatInfo,
Character
```

This structure of documents is illustrated in Figure 4.

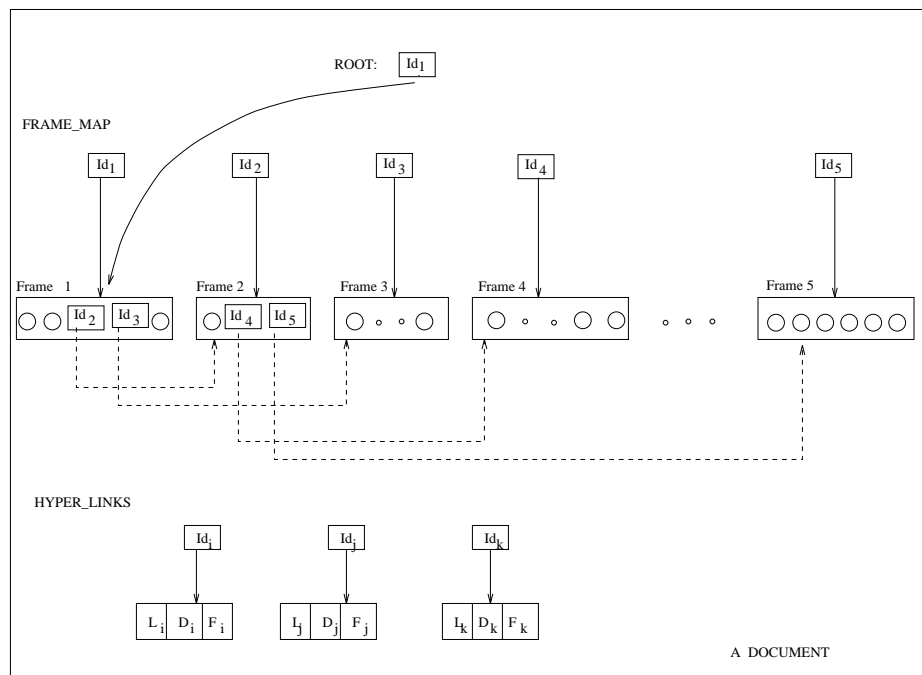


Figure 4: Structure of a Document

3.2 Consistency Conditions on a Document

As indicated in Section 3.1, the fields of the record type ‘Doc’ must satisfy various consistency conditions in order for it to represent only valid (or well-formed) documents. These conditions are:

1. the root of the document must occur in the domain of the frame map in order for the contents of the document to be defined;
2. the frame map must be non-circular to ensure that the contents of the document are finite. This is equivalent to the condition that no frame of the document should be contained in (a subframe of) itself;
3. any frame identifier which is defined as the source of some hyperlink in the document must belong to the document;
4. the frame map of the document must record all frames belonging to the document, including the root frame, and no others;
5. each frame in the document must have a unique identifier, that is a given frame identifier cannot appear in the contents of two different frames in the document. This is to ensure that it is possible to distinguish two frames in the document which have the same contents, for example when editing the document.

They are formalised in the function ‘is_wf_document’:

```

value
  is_wf_document : Doc → Bool
  is_wf_document(d) ≡
    let mk_Doc(r, fm, hm) = d in
      r ∈ dom fm ∧
      is_non_circular(fm) ∧
      dom hm ⊆ dom fm ∧
      let id_list = all_frames(r, fm) in
        is_non_repeating(id_list) ∧ (elems id_list = dom fm \ {r})
    end
  end

```

The definitions of the functions ‘is_non_circular’, ‘all_frames’ and ‘is_non_repeating’ which are used in the above definition are given in Section 3.3 along with the definitions of some other auxiliary functions.

We might also naively suppose that an additional consistency condition on libraries and hyperlinks is necessary to ensure that the library, document and frame identifiers appearing as the

destination of any hyperlink must all exist, that is that it must always be possible to follow any hyperlink to its destination. However, in a multi-user environment where different libraries are assumed to be owned by different users, this would effectively imply that a document appearing as the destination of some hyperlink can only be edited in such a way as to maintain the validity of that hyperlink. This is clearly unreasonable, particularly if the documents at the source and the destination of the hyperlink have different owners. The best we can do, therefore, is to allow a hyperlink to reference a non-existent destination but to provide a function for automatically removing such obsolete hyperlinks from a document automatically. These functions are defined amongst the general editing functions given in [4].

3.3 Auxiliary Functions

Here we give the definitions of the functions ‘is_non_circular’, ‘all_frames’ and ‘is_non_repeating’ which are used in the definition of the function ‘is_wf_document’ in Section 3.2, as well as the definitions of other auxiliary functions used in the definitions of these functions.

The function ‘string’ simply concatenates the lists of tokens in some list of pieces of formatted text into a single list of tokens, and the function ‘all_string’ extends this to act on a frame, returning a single list containing all the tokens in all the pieces of formatted text in that frame.

```

value
  string : FormattedText* → Token*
  string(ft) ≡
    if ft = ⟨⟩ then ⟨⟩ else chars(hd ft) ^ string(tl ft) end,

  all_string : Frame → Token*
  all_string(f) ≡ string(string(f))

```

The function ‘subframes’ filters a list of tokens, retaining only those which are frame identifiers.

```

value
  subframes : Token* → FrameId*
  subframes(s) ≡
    if s = ⟨⟩ then
      ⟨⟩
    else
      let t = subframes(tl s) in
        case hd s of Token_from_FrameId(id) → ⟨id⟩ ^ t, _ → t end
      end
    end

```

We use these functions to define the function ‘subframes_list’, which returns a list of the identifiers of all embedded subframes of a frame, and from that the function ‘all_subframes_list’, which returns a list of the identifiers of all subframes, both embedded and free, of a frame.

```

value
  subframes_list : Frame → FrameId*
  subframes_list(f) ≡ subframes(all_string(f)),

  all_subframes_list : Frame → FrameId*
  all_subframes_list(f) ≡ subframes_list(f) ^ freeFrames(f)

```

The function ‘is_subframe_of’ checks whether one given frame is a subframe, recursively and either embedded or free, of another. This is the case if the first frame is an immediate subframe of the second or if it is recursively a subframe of some immediate subframe of the second.

```

value
  is_subframe_of : FrameId × FrameId × (FrameId  $\xrightarrow{m}$  Frame) → Bool
  is_subframe_of(f1, f2, m) ≡
    f2 ∈ dom m ∧
    let l = all_subframes_list(m(f2)) in
      f1 ∈ elems l ∨
      ∃ f3 : FrameId • f3 ∈ elems l ∧ is_subframe_of(f1, f3, m)
  end

```

A frame map is then non-circular if it contains no frame which is recursively a subframe of itself.

```

value
  is_non_circular : (FrameId  $\xrightarrow{m}$  Frame) → Bool
  is_non_circular(m) ≡
    ∀ f : FrameId • f ∈ dom m ⇒ ~ is_subframe_of(f, f, m)

```

The function ‘all_frames’ returns a list of the identifiers of all the frames which are either embedded subframes or free frames of some given frame, both immediately and recursively. This first of all determines the immediate subframes of the given frame using the function ‘all_subframes_list’, then calls the function ‘frames_in_list’ on this list of frames and concatenates the two results.

The function ‘frames_in_list’ recursively determines a list of the identifiers of all frames contained in any of the elements of the given list of frame identifiers.

Each of these functions has a precondition which states that the frame map must be non-circular. This is to ensure that the calculation terminates: in a circular frame map some frame can be a subframe of itself.

```

value
  all_frames : FrameId × (FrameId  $\xrightarrow{m}$  Frame)  $\xrightarrow{\sim}$  FrameId*
  all_frames(f, m)  $\equiv$ 
    let
      l = if f  $\in$  dom m then all_subframes_list(m(f)) else  $\langle \rangle$  end
    in
      l  $\wedge$  frames_in_list(l, m)
    end
  pre is_non_circular(m),

  frames_in_list : FrameId* × (FrameId  $\xrightarrow{m}$  Frame)  $\xrightarrow{\sim}$  FrameId*
  frames_in_list(fl, m)  $\equiv$ 
    if fl =  $\langle \rangle$  then
       $\langle \rangle$ 
    else
      let  $\langle h \rangle \wedge t = fl$  in all_frames(h, m)  $\wedge$  frames_in_list(t, m) end
    end
  pre is_non_circular(m)

```

Finally, the function ‘is_non_repeating’ checks whether or not a given list of frame identifiers contains any duplicates:

```

value
  is_non_repeating : FrameId*  $\rightarrow$  Bool
  is_non_repeating(fl)  $\equiv$  len fl = card elems fl

```

4 Positioning within a Document

We now extend the model by introducing the notion of an *index* to define the abstract “position” of characters and frame identifiers within a document. This is particularly useful in the specification of editing operations on documents (see [4]). For example, we can give a single index to define the point at which new text is to be added to a document (this index representing something like the current position of the cursor in typical computer systems) or two indexes to define a region of text which is to be deleted from a document.

We represent a position as an index using a product type which consists of a frame identifier and two positive integers (the type ‘Index’ below). The type ‘Nat₁’ represents the positive integers as a subtype of the natural numbers.

```

type
  Nat1 = { | n : Nat • n  $\neq$  0 | },
  Index = FrameId × Nat1 × Nat1

```

The frame identifier in an index identifies the particular frame within which the indexed position is located, and the two positive integers fix the position within the contents of that frame. If the first integer is less than or equal to the number of pieces of formatted text the frame contains then it determines a particular piece of formatted text, namely the one at that numerical position in the list, and the second integer in turn defines a similar position in the list of tokens within that piece of formatted text. If the first integer is one greater than the number of pieces of formatted text contained in the frame then it refers to the list of free frames of the frame, and the second integer then determines a particular free frame in that list of free frames.

An index is said to be *valid* if it refers to the position of some character or frame identifier within the document, that is if the frame identifier in the index refers to some frame in the document and if the two positive integers define some valid location within the contents of that frame according to the interpretation given above. The two functions ‘is_valid_index’ define this property.

value

```
is_valid_index : Document × Index → Bool
is_valid_index(d, (f, n1, n2)) ≡
  let m = frame_map(d) in
    f ∈ dom m ∧ is_valid_index(m(f), n1, n2)
  end,
```

```
is_valid_index : Frame × Nat1 × Nat1 → Bool
is_valid_index(f, n1, n2) ≡
  let mk_Frame(or, st, ff) = f, l = len st in
    n1 ≤ l ∧ n2 ≤ len chars(st(n1)) ∨ n1 = l + 1 ∧ n2 ≤ len ff
  end
```

Using this definition of indexes and their validity, the particular token situated at some given valid index position is then found using the function ‘token_at_index’. This function applied to a frame simply returns the token at the given position in the appropriate piece of formatted text or in the free frames according to the values of the integers in the index as explained above.

value

```
token_at_index : Document × Index  $\xrightarrow{\sim}$  Token
token_at_index(d, (f, n1, n2)) ≡
  let m = frame_map(d) in token_at_index(m(f), n1, n2) end
  pre is_valid_index(d, (f, n1, n2)),
```

```
token_at_index : Frame × Nat1 × Nat1  $\xrightarrow{\sim}$  Token
token_at_index(f, n1, n2) ≡
  let mk_Frame(or, st, ff) = f, l = len st in
    if n1 ≤ l then chars(st(n1))(n2) else ff(n2) end
  end
  pre is_valid_index(f, n1, n2)
```

The function ‘next_index’ returns the index of the position immediately following some given valid current index. This function is not defined if the current index is positioned at the last token of the document (the function ‘is_end_doc’ in the precondition).

If the token at the current index position is a frame, then the next index is the first token in the first piece of formatted text inside that frame or, if the frame contains no pieces of formatted text, the first free frame inside that frame. If this frame contains neither formatted text nor free frames (i.e. is empty) or if the token at the current index position is a character then the next index is simply the index of the next token inside the current frame, except that if the current index is already positioned at the last token of the current frame then we regress to the position of the current frame within its containing frame and move to the next index from that. The auxiliary function ‘inc_index’ is used to calculate the next index position in this case.

value

```

next_index : Document × Index  $\xrightarrow{\sim}$  Index
next_index(d, id)  $\equiv$ 
  let (f, n1, n2) = id in
    case token_at_index(d, id) of
      Token_from_FrameId(new)  $\rightarrow$ 
        let i = (new, 1, 1) in
          if is_valid_index(d, i) then i else inc_index(d, id) end
        end,
      _  $\rightarrow$  inc_index(d, id)
    end
  end
pre is_valid_index(d, id)  $\wedge$   $\sim$  is_end_doc(d, id),

inc_index : Document × Index  $\xrightarrow{\sim}$  Index
inc_index(d, id)  $\equiv$ 
  let (f, n1, n2) = id, i1 = (f, n1, n2 + 1), i2 = (f, n1 + 1, 1) in
    if is_valid_index(d, i1) then
      i1
    elsif is_valid_index(d, i2) then
      i2
    else
      inc_index(d, index_of_frame(d, f))
    end
  end
pre is_valid_index(d, id)  $\wedge$   $\sim$  is_end_doc(d, id)

```

The auxiliary functions ‘is_end_doc’ and ‘index_of_frame’ which are used in the above functions are defined in Section 4.1 below.

4.1 Auxiliary Functions

The function ‘`index_of_frame`’ returns the index representing the position of a given frame identifier belonging to the contents of some document. This is defined implicitly in terms of the functions ‘`token_at_index`’ and ‘`is_valid_index`’ – the index required must be a valid index in the document and the token at that index position must be the given frame identifier. This is guaranteed to produce a unique result because of the constraint that any frame identifier cannot occur more than once in the contents of any given document.

Note that this function does not produce a result in the case where the given frame identifier is that of the root frame of the document because the root frame does not have a valid index according to our definition of indexes. This is not a problem, and in particular does not mean that we can never add any text to a new empty document because we cannot define the position at which it is to be added, because the editing and creation operations we define in [4] include an operation for adding a token to an empty frame and this can of course be applied to the empty root frame of an empty document.

value

```

index_of_frame : Document × FrameId  $\xrightarrow{\sim}$  Index
index_of_frame(d, f) as i
  post is_valid_index(d, i)  $\wedge$  token_at_index(d, i) = f
  pre f  $\in$  dom frame_map(d)  $\wedge$  f  $\neq$  root(d)

```

The function ‘`is_end_doc`’ checks whether a given valid index is the index of the last token of a document. If the token at the given index position is the frame identifier of some non-empty frame then this is not the end of the document because everything within the contents of this frame occurs at later positions. Similarly, if we are not at the end of the contents of the current frame then we are also not at the end of the document. If we are at the end of the contents of the current frame, then we are at the end of the document if the current frame is the root frame, otherwise we must regress to the position of the current frame within its containing frame and check whether or not we are at the end of that frame. This is done using the function ‘`check`’.

The function ‘`is_empty`’ checks whether or not a frame is empty. An empty frame has no formatted text and no free frames.

value

```

is_end_doc : Document × Index  $\xrightarrow{\sim}$  Bool
is_end_doc(d, id)  $\equiv$ 
  case token_at_index(d, id) of
    Token_from_FrameId(i)  $\rightarrow$ 
      if is_empty(frame_map(d)(i)) then check(d, id) else false end,
    _  $\rightarrow$  check(d, id)
  end
  pre is_valid_index(d, id),

```

```

check : Document × Index  $\xrightarrow{\sim}$  Bool
check(d, id)  $\equiv$ 
  let (f, n1, n2) = id, i1 = (f, n1, n2 + 1), i2 = (f, n1 + 1, 1) in
    if is_valid_index(d, i1)  $\vee$  is_valid_index(d, i2) then
      false
    elseif f = root(d) then
      true
    else
      check(d, index_of_frame(d, f))
    end
  end
pre is_valid_index(d, id),

is_empty : Frame  $\rightarrow$  Bool
is_empty(f)  $\equiv$ 
  let mk_Frame(⟦, st, ff) = f in st =  $\langle \rangle$   $\wedge$  ff =  $\langle \rangle$  end

```

5 Some General Auxiliary Functions

In this section we give some general auxiliary functions on documents and frames which are useful in the specifications given in the companion papers [5, 4].

The function ‘is_subframe_in’ checks whether a given frame is an embedded subframe of a second given frame in a given document.

```

value
is_subframe_in : FrameId × FrameId × Document  $\rightarrow$  Bool
is_subframe_in(i, j, d)  $\equiv$ 
  let fm = frame_map(d) in
    j  $\in$  dom fm  $\wedge$  i  $\in$  elems subframes_list(fm(j))
  end

```

The functions ‘freeframes_set’ and ‘subframes_set’ return respectively the set of all free frames and the set of all embedded subframes in a document. The second of these is calculated from the first using the function ‘all_frames’ which returns a list of the identifiers of all the frames which are either subframes or free frames in a document. The first is calculated using the auxiliary function ‘all_freeframes’ which returns a list of the frame identifiers of all the free frames in a document. This is in turn calculated with the help of the auxiliary function ‘freeframes_in_list’ which simply concatenates the lists of free frames of each frame in a given list of frame identifiers.

```

value
  freeframes_set : Document → FrameId-set
  freeframes_set(d) ≡ elems all_freeframes(d),

  subframes_set : Document → FrameId-set
  subframes_set(d) ≡ elems all_frames(d) \ freeframes_set(d),

  all_frames : Document → FrameId*
  all_frames(d) ≡ all_frames(root(d), frame_map(d)),

  all_freeframes : Document → FrameId*
  all_freeframes(d) ≡
    freeframes_in_list(all_frames(d) ^ ⟨root(d)⟩, d),

  freeframes_in_list : FrameId* × Document  $\xrightarrow{\sim}$  FrameId*
  freeframes_in_list(l, d) ≡
    if l = ⟨⟩ then
      ⟨⟩
    else
      freeFrames(frame_map(d)(hd l)) ^ freeframes_in_list(tl l, d)
    end
  pre elems l ⊆ dom frame_map(d)

```

Finally, the function ‘main_frame’ returns the root frame of a document, and the function ‘is_character’ simply tests whether a given token is a character.

```

value
  main_frame : Document → Frame
  main_frame(d) ≡ frame_map(d)(root(d)),

  is_character : Token → Bool
  is_character(t) ≡
    case t of Token_from_FrameId(_) → false, _ → true end

```

6 Conclusion

Based on an extensive analysis of multi-lingual documents, we have presented a very general model which can be used to describe multi-lingual documents in which the reading and writing direction of different parts of a document may be different. The model is based on a simple structuring of a document into frames. Frames can be arbitrarily nested, thus allowing arbitrarily

complex document structures, and the reading and writing direction is fixed as part of the parameters of the frame and does not change in any frame except perhaps within other frames inside it.

We have also extended the model by introducing the concept of an index, with which the position of any character or frame (identifier) within the contents of a document can be uniquely referenced, and by defining some general auxiliary functions which determine various properties of documents and frames.

Further extensions, covering the display and printing and the creation and editing of documents within the model, can be found in the companion papers [5, 4].

Acknowledgments

Many thanks to Chris George for reviewing the manuscript and providing helpful comments. Myatav Erdenechimeg thanks UNU/IIST for the use of their facilities during the course of this work.

References

- [1] Peter T. Daniels and William Bright, editors. *The World's Writing Systems*. Oxford University Press, 1996.
- [2] Myatav Erdenechimeg and Richard Moore. Multi-directional Multi-lingual Script Processing. Technical Report 75, UNU/IIST, P.O.Box 3058, Macau, June 1996.
- [3] Myatav Erdenechimeg and Richard Moore. Multi-directional Multi-lingual Script Processing. In *Proceedings of the Seventeenth International Conference on the Computer Processing of Oriental Languages, Vol. 1*, pages 29 – 34. Oriental Languages Computer Society, Inc., 1997.
- [4] Myatav Erdenechimeg and Richard Moore. MultiScript III: Creating and Editing Multi-lingual Documents. Technical Report 113, UNU/IIST, P.O.Box 3058, Macau, September 1997.
- [5] Yumbayar Namsrai and Richard Moore. MultiScript II: Displaying and Printing Multi-lingual Documents. Technical Report 112, UNU/IIST, P.O.Box 3058, Macau, June 1997.
- [6] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.
- [7] I. J. Schmidt. *Grammatik der mongolischen Sprache*. 1831.