Supporting the user's design of alternative complex notations

Silas Brown and Peter Robinson

University of Cambridge Computer Laboratory
15 JJ Thompson Avenue, Cambridge CB3 0FD, UK
{Silas.Brown,Peter.Robinson}@cl.cam.ac.uk

Abstract

One way to facilitate the accessibility of complex documents is to transcribe them into notations that are more suited to presentation on the user's preferred output device(s), whether they are visual or not. Users sometimes prefer to design their own notations for this purpose, particularly in the educational subjects that have no widely-used standard for performing such transcription. We have developed a software framework that can assist users to design their own complex notations; it can also be used for assisting the development of converters for standard notations. The framework is domain-independent, but we demonstrate it by giving examples of its use in mathematics and diagrams, using both visual and non-visual interaction.

1 Introduction

Educational and cultural activities make use of many complex notations beside simple text. Well-known examples include musical scores, mathematics, schematics, and scientific models. It is also possible to observe the use of non-standard, invented notations, particularly during activities that involve analysis for academic or training purposes, in which a document in one notation is annotated using another notation. However, invented notations have a broader potential, since in some cases they can be used to address circumstances in which someone who has difficulty using a standard notation may be able to use an alternative notation or an invented notation. Examples of such circumstances include various combinations of:

1. Visual difficulties. In some cases there are standard notations that can be used by blind people, such as various standards for Braille music and mathematics. However, not everyone with visual difficulties is totally blind, and many work more efficiently using their residual vision and may wish to design or customise notations to suit their individual visual profiles [8,9]. Notations can be customised for different tasks, such as sequential reading, rapid overview, or detailed analysis. Often it is desirable to omit or include certain details depending on how the document will be used, because people with print disabilities are frequently unable to skip over unwanted information at speed.
2. Dyslexia. Gregor and Newell's experiments [7] showed that allowing dyslexic users greater control over the presentation of information can help in some cases. This is logically extensible to complex notations.
3. Physical and hand-eye co-ordination difficulties that affect a person's ability to input a notation. Allowing the person to invent a new notation that he or she can input more easily can help to alleviate this. Optimising for input and/or editing is different from optimising for reading. Even direct-manipulation interfaces sometimes use a hidden input notation in their controls, although there is usually some compromise so that the input and output notations are conceptually similar. This need for similarity can be overshadowed by a disabled user's accessibility needs.
4. The use of devices such as mobile computers, which can exacerbate these problems.

The use of unusual notation systems is an under-explored option in the relief of difficulties, primarily because present-day computer software does not offer enough support for the creation of new notations. We have developed a generalised transformation framework that can be applied to many different transformations, particularly within the area of converting notations to cater for special difficulties. To support the creation of new notations, our transformation system addresses the following high-level goals:

1. The method of programming the framework for a particular type of conversion should encourage, wherever possible, a consideration of the notations themselves rather than the algorithmic methods for their transformation.
2. Prototyping and customising new transformation tasks should require as little effort as possible.

This paper describes our framework and gives examples that illustrate its potential.

2 Related work

Specialist transformation tools are already in use to aid accessibility. For example, AsTeR [13] is a comprehensive system for reading mathematical documents, either interactively or non-interactively, as speech along with background sounds and voice modulations. Also of note is software that produces specialist Braille notations, such as the Braille music systems MFB [11] and Goodfeel [12], and the Web Access Gateway [2]. These systems are not generalisable; the transformations they employ are limited to a given domain such as mathematics or music, and, although they might permit some customisation, they cannot easily be programmed to handle completely new transformation tasks, even among the many different standards and house styles that are associated with the notations they are designed to work with. The problem is further complicated by the fact that the source material is stored in many diverse formats; multiple conversions are often required, sometimes leading to information loss due to the limitations of intermediate formats.

There are also generalised, programmable transformation frameworks such as TXL [4] and XSLT [14], which can be used as programmers' tools to implement new transformation tasks as needed. These systems deal with generalities, such as symbols and data, that can apply to many different types of notation. Although any programming language can be used for this, tools aim to assist by supporting particular design approaches that are appropriate for transformation tasks. However, since these tools are not primarily designed for the conversion and adaptation of the notations of various educational disciplines, the design approaches that they encourage are not necessarily the most appropriate; in particular, they tend to encourage a focus on algorithms rather than on the notations themselves.

To illustrate this, consider the case of rewriting-based systems such as TXL, DMS, Stratego, Mathematica, Rigal, Prolog and Gupta et al's logical denotations. Rewriting works best for mathematical structures where the rewriting rules follow naturally from the mathematical definition of the structure. Any transformation that can be expressed informally as a set of "this pattern should be re-written as that" statements, which capture the complete transformation and are not merely examples of it, is likely to be easily implemented in a rewriting system so long as the input data can be parsed into the system. However, rewriting systems are more difficult to use in cases where it is less obvious what the rules should be, or when the transformation needs to go through one or more intermediate stages before the desired result can be achieved; this needs more thought on the part of the transformation programmer. In particular, when a single hierarchical structure is not the most natural way to represent the structure of a notation, the additional transformation between the structure in which the notation is stored and that in which the desired transformation is most clearly expressed can impose artificial overhead on the transformation code.

3 Use of 4DML in designing new notations

Our transformation system 4DML supports the user’s design of alternative complex notations by encouraging a consideration of the notations themselves rather than the algorithmic methods for their transformation. It allows the user to specify the structure of the desired result in a fairly concise manner as a 4DML “model”, without focusing on algorithms or transformation rules as is normally the case. This is not to say that a 4DML model does not specify an algorithm. It does specify an algorithm, because the behaviour of models is well-defined. 4DML does not “guess” the user’s intentions, but uses the model as a guide to reading and re-structuring the input in a distinct manner. Authors of models know what they can expect 4DML to do; it is not necessary to “train” the system with many examples, and there is no uncertainty as to whether or not it will “understand”. However, the form of 4DML models is such that it puts the emphasis on the desired output notation rather than on the process. Hence 4DML encourages a consideration of the notations themselves without introducing the overhead of training by example. The model language also aims for a balance between brevity and readability and is not unnecessarily verbose, which helps with rapid or experimental prototyping when designing new notations.

Our transformation utility converts data from XML or another source into 4DML, and then outputs it in the order dictated by a “model”, an example of which is shown in Figure 3. The intermediate data structure that can represent both tree-like and matrix-like structures in any number of dimensions and supports multiple independent structures over the same data. Figure 4 shows a representation of the 4DML data structure. In Figure 4, the text in each coloured box corresponds to the name of an element; the numerical subscript (which is also represented by the box's colour) shows the position of that element among its peers; the depth of the element in the tree is shown by the vertical position of the box, and the data symbols that are enclosed in these elements are shown along the horizontal axis (each symbol is uniquely identified where necessary). The left-to-right order of the columns is not important, since it can be re-constructed from the numbers when necessary; otherwise, 4DML is free to sort the symbols into a different order as required by the output notation. The absence of a "global" left-to-right ordering helps with representing multiple independent sets of markup over the same data, since the order in which the data is to be read might depend on which set of markup is in use. So the ordering should be a property of the markup, not of the underlying data; the markup is effectively a system for indexing into the data.

4DML has been described in detail elsewhere [3,1]; this paper aims to demonstrate by example some of the possibilities for its use in designing new notations. An individual with low vision has used the system for the tasks described in this section.

3.1 Mathematics

4DML was used to parse an HTML document that contained mathematical notation in MathML, and to output the result in several formats including speech, Braille, and newly-invented mathematical notations suitable for low vision.

For example, the expression $\sum_{n=0}^{k} \frac{f^n a}{n}$, which in MathML markup is

```
    <math xmlns="&mmlns;" mode="inline">
      <msubsup>
        <mo>&#x2211;</mo>
        <mrow> <mi>n</mi> <mo>=</mo> <mn>0</mn> </mrow>
        <mrow> <mi>k</mi> </mrow>
      </msubsup>
      <mfrac>
        <mrow>
          <msup>
            <mi>f</mi>
            <mrow> <mi>n</mi> </mrow>
          </msup>
          <mi>a</mi>
        </mrow>
        <mrow> <mi>n</mi> </mrow>
      </mfrac>
    </math>
```

became in speech "sigma from n equals 0 to k of f to the n a over n". In this case no unusual transpositions need to be done; the transformation works sequentially using top-down recursion. In XSLT, one would write code such as the following (in each case, only an extract from the complete transformation will be shown, due to the large number of diverse mathematical symbols that it must handle):

```
<xsl:template match="mo">
  <xsl:variable name="cdata" select="fn:string()"/>
  <xsl:if test="eq($cdata,'&#0x2200')">
    <xsl:text>for all</xsl:text>
  </xsl:if>
</xsl:template>

<xsl:template match="mfrac">
  <xsl:apply-templates select="child::node()[1]"/>
  <xsl:text> over </xsl:text>
  <xsl:apply-templates select="child::node()[2]"/>
</xsl:template>
```

meaning “find any `mo` elements (`mo` is the MathML code for Mathematical Operator) that contain the entity code `&#x2200;` (which represents ∀), and output `for all` for any that are found” and “for any `mfrac` (MathML fraction) elements, process the first child, output `over` and process the second child”. The rewriting-based language TXL is somewhat clearer than XSLT in this case:

```
rule convert_forall
  replace [element]
    <mo> &#x2200 </mo>
  by ( for all )
end rule

rule convert_mfrac
  replace [element]
    <mfrac> A [element] B [element] </mfrac>
  by ( A over B )
end rule
```

(The above code assumes the presence of TXL code to parse XML and to represent output character data, and a main rule that invokes the rules appropriately; this runs to several hundred lines.)

The equivalent in 4DML’s Compact Model Language is arguably more concise than both of the above:

```
mo value="&#x2200" / "for all"

mfrac / (
      anyName number=1 call=math,
      " over ",
      anyName number=2 call=math
)
```

Because of this reduced complexity, developing in 4DML rather than in XSLT facilitates the rapid experimentation of new types of output notation. For example, by changing the 4DML model, Braille output can be produced (Figure 1), in this case using the Nemeth code [5] but it is possible to use other codes. Here the Braille is in the form of annotated TEX graphics so that a sighted person can see how Braille mathematics works, but it can also be output as codes suitable for controlling an automated Braille embosser or a Braille display.

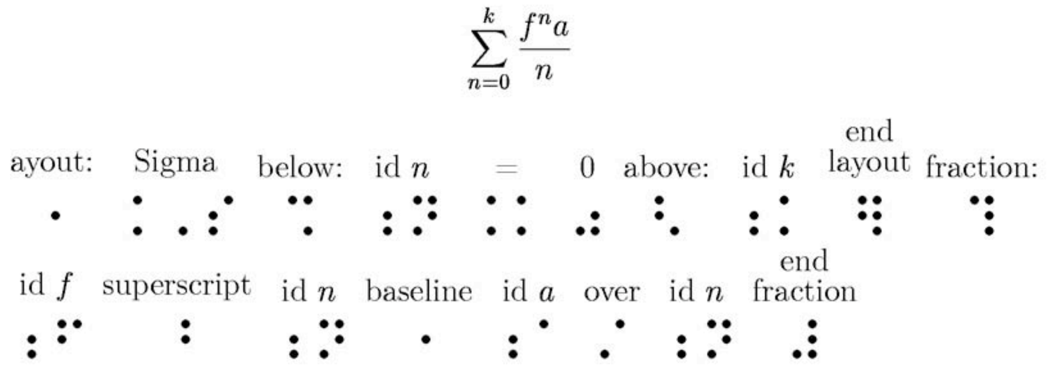


Figure 1: Annotated Braille mathematics (Nemeth linear code)

The model can also be adjusted for invented mathematical notations, such as those shown in Figure 2. In this case the output was achieved by driving the layout engine Lout [10], which is a general document preparation system that takes a description of page layout and typesets it as PostScript or PDF. Lout’s language has constructs for coloured boxes and other shapes, and these can be inserted in the 4DML model in various ways to experiment with new notations. The exact nature of the desired notation will depend on how well the user can work with colour, with nonalphabetic symbols, with smaller print sizes (superscripts and subscripts), and with spatial layouts; aspects of presentation that the user has difficulty with can sometimes be compensated for by other aspects of presentation that are not normally used but that the user finds more convenient.

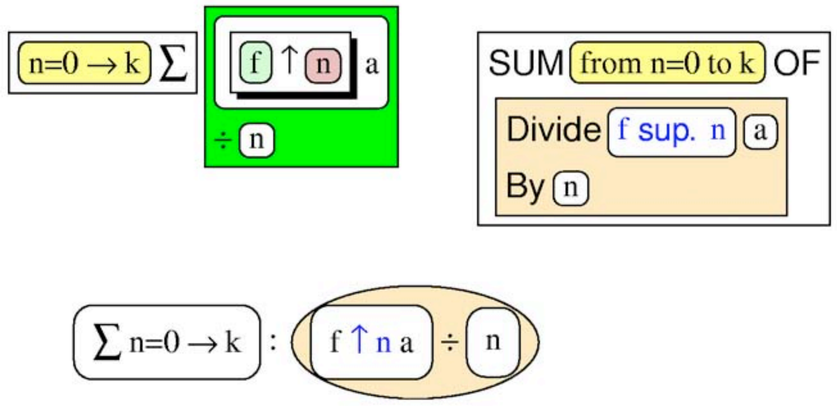
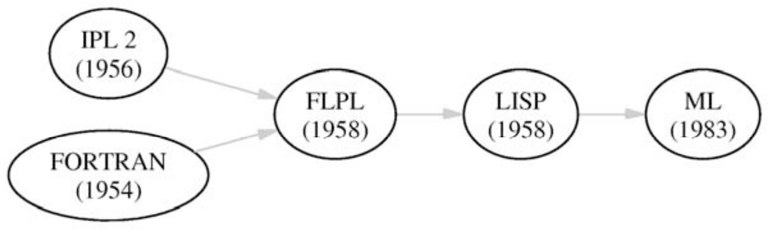


Figure 2: Some invented mathematical notations

Mathematical notation does not fully demonstrate the novelty of 4DML because mathematics can already be transformed and customised using existing specialised systems and generalised rewriting-based systems. 4DML's novelty is better shown in other domains, but we felt it was important for any new transformation system to show that it can process mathematics as well.

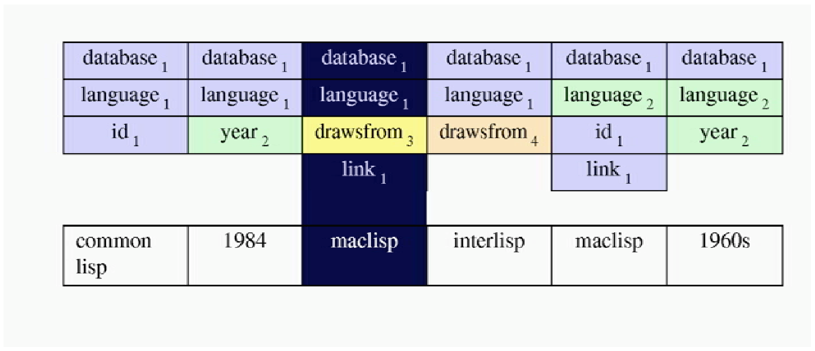
3.2 Diagrams

4DML was used to transform an XML database of some programming languages and their historical relationships into input suitable for AT&T GraphViz [6] to draw a directed graph. A separate model was used to derive a subset of the database suitable for producing a graph that could fit in a limited space without reducing the print size (Figure 3). The model specifies “ML” as a starting point and instructs 4DML to find languages that had influenced it, using automatically-generated link elements that connect any duplicate strings (the text link is specified using a 4DML command-line switch). The method of traversing links using 4DML’s `broaden` directive is illustrated in Figure 4; it does not depend on a particular link direction, and the same method can be used for more complex relationships.

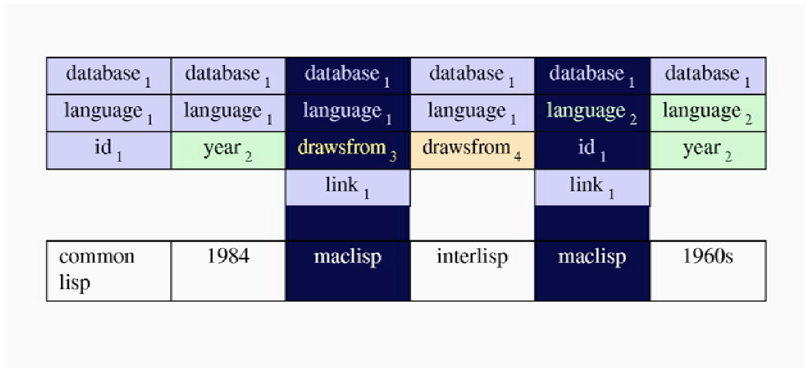


```
digraph PLFT { rankdir="LR"
[[cml library export-code/
   doANode/(]]
  [[cml id]] [label="[[cml name]]\n([[cml year]])"];
  [[cml drawsfrom/(link, " -> ", id)]];
  [[cml drawsfrom/link broaden/id/language broaden
  call=doANode )
 ),
language no-strip/id value=ml/language broaden call=doANode
]]
}
```

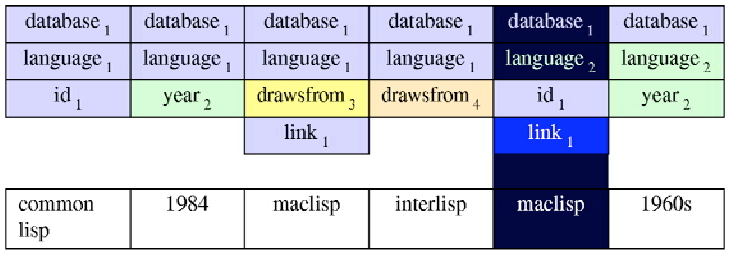
Figure 3: Subset of a database of historical relationships between programming languages, extracted by 4DML and typeset by GraphViz, and the 4DML model that produced it. 4DML code is enclosed with `[[cml...]]` and everything else is GraphViz code.


database 1 database 1 database 1 database 1 database 1 database 1
language 1 language 1 language 1 language 1 language 2 language 2
id 1 year 2 drawsfrom 3 drawsfrom 4 id 1 year 2
link 1 link 1
common lisp 1984 maclisp interlisp maclisp 1960s

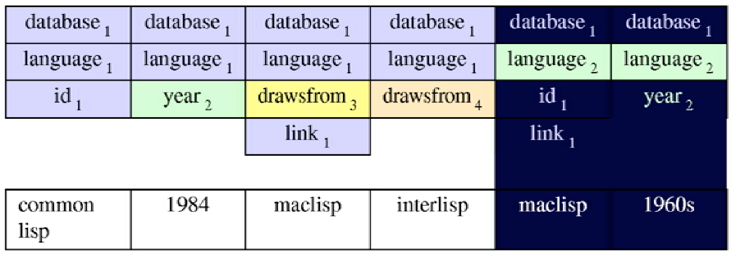
(a) `drawsfrom`


database 1 database 1 database 1 database 1 database 1 database 1
language 1 language 1 language 1 language 1 language 2 language 2
id 1 year 2 drawsfrom 3 drawsfrom 4 id 1 year 2
link 1 link 1
common lisp 1984 maclisp interlisp maclisp 1960s

(b) `link broaden`


database 1 database 1 database 1 database 1 database 1 database 1
language 1 language 1 language 1 language 1 language 2 language 2
id 1 year 2 drawsfrom 3 drawsfrom 4 id 1 year 2
link 1 link 1
common lisp 1984 maclisp interlisp maclisp 1960s

(c) `id`


database 1 database 1 database 1 database 1 database 1 database 1
language 1 language 1 language 1 language 1 language 2 language 2
id 1 year 2 drawsfrom 3 drawsfrom 4 id 1 year 2
link 1 link 1
common lisp 1984 maclisp interlisp maclisp 1960s

(d) `language broaden`

Figure 4: Link traversal in 4DML

To understand how the model in Figure 4 operates, first consider Figure 4a. It shows a representation of 4DML's internal format, holding some data about the historical relationships between some programming languages which was read from an XML file. There is an implicit link between the two instances of “maclisp”, which is made explicit by the automatic addition of a “link” element (uniquely numbered) covering both ends of the link. In Figure 4a, the 4DML model specified “drawsfrom”, meaning that each element or attribute named “drawsfrom” should be selected (because of the surrounding parts of the model, this is done within the scope of the currently-selected “language” element). In Figure 4b, the link is selected and “broadened”, so that both ends of it are in scope. Figure 4c selects which end is of interest (in this case, the object that is being referenced by the link, but it could just as easily have been other parts of the data that also make reference to that object) and Figure 4d broadens the scope of this to the larger amount of data that is of interest (in this case, all of the details of the programming language that is being linked to). Note that this method of addressing the data breaks free of the hierarchical constraints as necessary, and it can be expressed very concisely in 4DML’s model language.

By comparison, consider how complex this would be to achieve in a rewriting language such as TXL. Aside from constructing a grammar for XML and for the output language, the data will have to be re-ordered in several steps to simulate the effects of the link traversal, using TXL code such as:

```
rule traverse_links
  replace [repeat element] N
  construct NewN [repeat element] N [init] [make_closure] [elim_other_langs]
end rule

function init
replace [repeat element]
  Before [repeat element]
  <language> R1 [repeat element]
    <id> ML </id> R2 [repeat element]
  </language>
  After [repeat element]
by
  <language-to-ouput> <id>ML</id> R1 R2 </language-to-output>
  Before After
end function

function make_closure
replace [repeat element]
  Before [repeat element]
  <language-to-output> A [repeat element]
  <drawsfrom> L </drawsfrom> B [repeat element]
  </language-to-output>
  Between [repeat element]
  <language> C1 [repeat element]
  <id> L </id> C2 [repeat element]
  </language>
  After [repeat element]
by
  Before
  <language-to-output> A
  <drawsfrom> L </drawsfrom> B
  </language-to-output>
  <language-to-output> <id> L </id> C1 C2
  </language-to-output>
  Between After
end function

function elim_other_langs
replace [element]
  <language> A [repeat element] </language>
```

```
by
end function
```

The child-elements of the individual elements will then have to be sorted into a normalised order (which is almost as much code again), and then transformed using code such as the following:

```
function transform
  replace [element]
    <language-to-output>
      <id> I [repeat element] </id>
      <name> N [repeat element] </name>
      <year> Y [repeat element] </year>
      D [repeat element]
    </language-to-output>
  construct NewD [repeat element] I D [handle_drawsfrom]
  by
    I label= N ( Y ) ; D
end function

function handle_drawsfrom
  replace [repeat element]
    <id> ThisID [repeat element] </id>
    <drawsfrom> OtherID [repeat element] </drawsfrom>
    R [repeat element]
  construct NewR [repeat element] ThisID R [handle_drawsfrom]
  by
    OtherID -> ThisID ; NewR
end function
```

In reality, the TXL code is still more complicated due to the need to wrap many of the output symbols as indirect tokens so that they are not confused with TXL operators.

XSLT code can be briefer than the TXL in this case, due to the ability to specify a behaviour similar to 4DML's "broaden" by using variables and XPATH (however, XSLT stylesheets often have to make more assumptions about the input structure than 4DML models do). In this case, the XSLT would include code such as:

```
<xsl:template match="language">
  <xsl:value-of select="id/fn:string()" />
  <xsl:text> [label="</xsl:text>
  <xsl:value-of select="name/fn:string()" />
  <xsl:text>\n(</xsl:text>
  <xsl:value-of select="year/fn:string()" />
  <xsl:text>)"];</xsl:text>
  <xsl:process-templates select="drawsfrom">
    <xsl:with-param name="linksTo" select="id/fn:string()" />
  </xsl:process-templates>
</xsl:template>

<xsl:template match="drawsfrom">
  <xsl:param name="linksTo" />
  <xsl:variable name="cdata" select="fn:string()"/>
  <xsl:value-of select="$cdata">
  <xsl:text> -> </xsl:text>
  <xsl:value-of select="$linksTo">
  <xsl:text> ; </xsl:text>
  <xsl:process-templates select="/language[eq(id/fn:string(),$cdata)]" />
</xsl:template>
```

However, it is clear that 4DML is the more concise option (Figure 3).

A modified version of the 4DML model produced text suitable for a speech synthesizer (Figure 5). In this way alternative ways of drawing or reading the diagram can be produced and experimented with.

SPEECH SYNTHESIZER: "ML (1983) draws from LISP. LISP (1958) draws from FLPL. FLPL (1958) draws from IPL 2 and FORTRAN. IPL 2 (1956). FORTRAN (1954)."

Figure 5: As Figure 3 but formatted for speech synthesis

4 Conclusion

This paper aimed to demonstrate that designing new notations and converting between them has potential to aid universal access to educational activities. It illustrated this by using our 4DML system to convert between different ways of writing mathematics and diagrams. We show elsewhere [3,1] how the same system can be applied to several different musical notations, to language analysis, and to supporting different ways of inputting these notations. 4DML allows one to experiment with new notations more easily because its model language is brief but readable and it encourages a consideration of notations rather than algorithms. We hope that distributors of electronic courseware will consider supporting the transformation of complex notations and allow the user to design his or her own notations, so that a much larger number of special circumstances can be accounted for.

References

[1] Silas S. Brown. Conversion of notations. Technical Report UCAM-CL-TR-591, University of Cambridge, Computer Laboratory, June 2004. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-591.pdf.

[2] Silas S. Brown and Peter Robinson. A World Wide Web mediator for users with low vision. In ACM CHI 2001 Workshop No. 14. http://www.ics.forth.gr/proj/at-hci/chi2001/files/brown.pdf.

[3] Silas S. Brown and Peter Robinson. Transformation frameworks and their relevance in universal design. Universal Access in the Information Society, 3(3-4):209-223, 2004.

[4] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In Proceedings of The International Conference of Computer Languages, pages 280-285, Miami, FL, Oct 1988.

[5] Abraham Nemeth et al. The Nemeth Braille Code for Mathematics and Science Notation. American Printing House for the Blind, 1972.

[6] E. R. Gansner, S. C. North, and K. P. Vo. DAG-a program to draw directed graphs. Software-Practice and Experience, 17(1):1047-1062, 1988.

[7] Peter Gregor and Alan F. Newell. An empirical investigation of ways in which some of the problems encountered by some dyslexics may be alleviated using computer techniques. In Proceedings of the Fourth International ACM Conference on Assistive Technologies ASSETS 2000, pages 85-91, Nov 2000.

[8] Julie A. Jacko, Max A. Dixon, Robert H. Rosa, Jr., Ingrid U. Scott, and Charles J. Pappas. Visual profiles: A critical component of universal access. In Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems, volume 1 of Profiles, Notes, and Surfaces, pages 330-337, 1999.

[9] Julie A. Jacko and Andrew Sears. Designing interfaces for an overlooked user group: Considering the visual profiles of partially sighted users. In Third Annual ACM Conference on Assistive Technologies, pages 75-77, 1998.

[10] Jeffrey H. Kingston. The design and implementation of the Lout document formatting language. Software-Practice and Experience, 23:1001-1041, 1993.

[11] Didier Langolff, Nadine Jessel, and Danny Levy. MFB (music for the blind): A software able to transcribe and create musical scores into Braille and to be used by blind persons. In Proceedings of the 6th ERCIM Workshop on `User Interfaces for All', number 17 in Short Papers, page 6. ERCIM, 2000.

[12] Bill McCann. GOODFEEL Braille Music Translator, Jun 1997. Dancing Dots Braille Music Technology, http://www.dancingdots.com/.

[13] T. V. Raman. Audio System for Technical Readings. PhD thesis, Cornell University, 1994.

[14] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, W3C Recommendation, Nov 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.