Visualizing Z notation in HTML documents

Paolo Ciancarini, Cecilia Mascolo, and Fabio Vitali

Department of Computer Science,
University of Bologna,
Mura Anteo Zamboni ,7, I-40127 Bologna (Italy)
tel: +39 51 354506, fax: +39 51 354510
e-mail: {ciancarini, mascolo, vitali}@cs.unibo.it

Abstract. The use of the WWW as a communication medium for software engineers is limited by the lack of tools for writing, sharing, and verifying formal notations. For instance, the Z specification language has a a rich set of mathematical characters, and requires graphic-rich boxes and schemas for its specifications. It is difficult to integrate Z specifications and text on WWW pages written with the current versions of HTML, and traditional tools are not suited for the task.
We present a Java-based tool for rendering Z specifications within HTML documents that can be shown on every WWW browser with Java capabilities. Being a complete rendering engine, text parts and Z specifications can be freely intermixed, and all the standard features of HTML (such as links, etc.) are available outside and inside Z specifications. Furthermore, the extensibility of our engine allows additional notations to be supported and integrated with current ones.

1 Introduction

The use of the WWW as an environment for software design introduces new problems and challenges to the specification community: the use of the WWW to support software process workflows, sharing specification documents, allowing read and write access, and providing hypertextual links among documents is a hot topic [8, 14].

The typographical rendering of WWW documents is usually defined using the HTML mark up language [1, 13]. HTML provides textual support for elements such as input fields, buttons, choice lists, etc. along with structural and formatting commands for text within the data format of network documents.

This has allowed both complex interfaces and proper and traditional text content to be described in ASCII-based source documents. On the other hand, HTML only allows a few elements, that is, those that are explicitly defined in the standard. Whenever authors' needs exceed the capabilities of the elements already defined in HTML, a different approach needs to be used: either the existing tags are abused for a different purpose than that for which they were created, or an image is used, or a Java applet is created providing the desired functionality.

Specification languages like Z [16] are often based on specialized notations (mathematics and logic symbols): it would be useful to be able to give a visual interpretation of these symbols and to allow them to be displayed on WWW pages.

The purpose of this paper is to report on a Java rendering engine for HTML data that we have implemented. The engine allows typographical and graphical support for standard HTML documents, and can easily be extended to include support for additional notations. We have added a complete graphical and typographical support for formal specification documents written in Z. The rendering engine we are describing is a completely autonomous piece of code that can work on unmodified Java-enabled browsers such as Netscape Communicator or Microsoft Internet Explorer.

Using our engine, it becomes possible to integrate full-blown Z specifications and plain text chunks using HTML and HTML-extensions that can be browsed and displayed in any WWW browser that is Java-enabled.

The paper is structured as follows: in Section 2 we summarize the state of the art of the rendering of Z documents as hypertexts. In Section 3 we describe the idea of extending HTML with Java using the *displets* concept. Section 5 explains how our Z browser works from the point of view of the user, whereas Section 4 describes the implementation details of the tool. In Section 6 we draw some conclusions and sketch our future work.

2 Creating Z specifications

2.1 Writing, printing and visualizing Z specifications

Several tools exist to this date to help software designers write, test, and share their Z specifications. A complete guide to all the existing tools for Z can be found at `http://www.comlab.ox.ac.uk/archive/z.html`.

We can divide these tools in four main categories: fonts, browsers, editors, and type checkers.

True Type fonts for Z are available to use with common word processors on many platforms including Windows and Macintosh, but fonts of course only give access to the special mathematical characters of the Z language, forcing users to use non-specific features of available tools to create the graphic boxes of schemata and other Z elements.

Customizable formatters such as LaTeX [10] are the most common tools to write Z specifications. General style files for LaTeX such as `oz.sty, fuzz.sty, ztc.sty`, have been published to precisely render Z specifications.

Logica has created a syntax-driven WYSIWYG editor for Z on MS Windows platforms. Such an editor also integrates a type checker and forces the production of well-formed Z specifications by providing facilities for building, editing, checking, and viewing Z specification documents. Being WYSIWYG, the editor can display the Z constructs and symbols as they would appear on a printed page.

Z Browser, an application for displaying Z specifications that runs on MS Windows, is presented in [12]. The tool is aimed at Z novices, and is integrated with a complete help system for Z grammar and notation.

Zola, by IST, is an integrated editing tool for Z that runs on Sun OS and Solaris. Zola is a WYSIWYG editor that automates the construction, syntax checking and visual layout of Z documents.

Several analysis tools also exist for Z specifications. For instance, CADiZ [7] is an integrated suite of tools for creating Z documents. It understands source files in LaTeX and Word for Windows, and can visualize implicit Z expressions (i.e. schema calculi) by showing their expansions.

Finally, the ZTC [18] type checker accepts LaTeX-formatted Z specifications as well as text-based ones. ZTC also suggests using a special syntax based on concatenation of ASCII characters for mathematical symbols.

2.2 Hypertext and Z specifications

There are several good reasons to provide hypertext functionalities to Z specifications. A complex specification is intrinsically composed of many connected chunks (schemas, etc.) that refer to each other in a peculiar, often unpredictable way. Furthermore, the idea of literate programming requires that schemas and texts interleave freely, so that the reader is provided with a narrative explanation of the most complex schemas, and a formalized and exact specification of vaguer descriptions. These remarks naturally call for a hypertext solution.

Moreover, collaboration and sharing are even better reasons for providing hypertext support to Z specifications: formal specifications are but one step in the complex process of system design, verification, and implementation [5]. Modern development processes are enacted by teams of people that cooperate, interact, and discuss. Being able to create, access, and verify formal specifications within the usual tools of our everyday work, publish them, connect them to the other deliverables of the design and implementation processes would allow a tighter integration between formal design and actual implementation [3].

Till recently, Z specifications could only be visualized on the WWW by creating images in one of the supported inline formats, such as GIF. This leads to a very cumbersome and unnatural creation process, since the Z specifications have to be created in a different environment than the text, and furthermore non-specialized graphic editors have to be used in order to produce graphically acceptable schemas. It is also a very unnatural and clumsy way of accessing the information: an image of a schema is a completely opaque object, where the subparts, the texts, and the formulas are completely unaccessible; it is a bitmap that cannot be further processed because the content and meaning have been lost: the content of a schema cannot be searched, the specifications cannot be indexed, analyzed or verified.

A first attempt to show Z specifications on the WWW was described in [11], designing a plug-in for Netscape and Internet Explorer that accepts Z specifications written using one of the existing LaTeX styles.

Although this approach is very original it has two main limitations: first, visualizing Z documents requires the availability of the plug-in, which is architecture-dependent (it only exists for MS Windows). Secondly, the LaTeX format is alien to the available SGML-based formats suggested for the WWW: in fact, writing Z schemas in LaTeX requires a different syntax and approach than writing the surrounding free-flow text in HTML, and the specifications live independently of the host document. The first problem has been addressed: the Z browser is becoming a Java applet, which is architecture-independent and can be run on most computers of the current generation.

We know that also J. Bowen and others in Reading are working on a Java applet to visualize Z schemas (`http://www.reading.ac.uk/s̃su95dlc/project/`). Our approach, detailed in section 5, is related but with noteworthy differences.

2.3 The advantages of markup languages

HTML has been extremely successful in allowing unsophisticated network users to become authors of fairly complex documents, even in the absence of widespread editing tools. Nonetheless, there has been in the past two or three years a widespread awareness ([15]) that HTML has reached its potential, and that a change of paradigm was necessary.

The major drawback of HTML is that it allows only a pre-specified set of elements. Authors can only use these elements, and have to limit their authoring needs to what is available within the existing language, or to force these elements beyond their intended meaning.

HTML is an application of the Standard Generalized Markup Language [15], that is, a class of documents conforming to the SGML Document Type Definition (DTD) that describes 'HTML documents'. SGML, being a meta-language describing classes of documents rather than one specific class, is free of the above mentioned limitations of HTML: by appropriately creating a custom class of documents, and defining the legal elements therein, authors can provide support for any kind of rhetorical need, however complex and arcane. Metalanguages 'allow groups of people or organizations to create their own customized mark up languages for exchanging information in their domain (music, chemistry, electronics, hill-walking, finance, surfing, linguistics, knitting, history, engineering, rabbit-keeping, etc. - see `http://www.ucc.ie/xml/`').

Unfortunately, SGML is considerably more complex to learn and use than HTML, and it has been said that this would prevent its generalized adoption. Therefore the SGML working group of the Word Wide Web Committee was asked to develop a new mark-up meta-language, namely the eXtended Markup Language (XML) [9], to take the place of SGML on the Web. XML documents would have to be straightforwardly usable over the Internet, compatible with SGML, and easy to create.

Interestingly, in the Z community an SGML-based language for Z specifications already exists: the Z Interchange Format (ZIF for short) [2] defines a portable representation of Z, that can be used by all tools supporting SGML. The ZIF is basically a Document Type Definition (DTD), namely an SGML specification defining the syntax of documents that contain Z specifications. In [6] a study of the usage of the ZIF was presented, according to which ZIF can be fruitfully used to create editors for Z documents using standard SGML tools, and that Z specifications encoded using ZIF could easily be included in other SGML documents.

XML documents are valid SGML documents. Most existing SGML DTDs can be used with no modifications in an XML environment. Notably, the Z Interchange Format is one of such DTDs.

It is therefore possible to use the definitions specified in the ZIF within XML tools, in order to create web-friendly visualizations of Z specifications. Alternatively, XML tools allow the HTML tag set to be described and extended as needed. By joining the HTML DTD with the ZIF DTD, and producing a capable browser, it is possible to write HTML documents that contain Z specifications as markup items, instead of images, thereby keeping all the useful properties that markup has over bitmaps.

In this paper we report about one such tool, that allow the display of standard HTML documents enriched with Z specifications, using a markup extension of HTML directly derived from ZIF.

3 Displets and HTML extensions

Displets were proposed in [17] as a way to extend HTML documents using Java. The HTML language was extended on a per-document basis by defining new tags as needed, and providing Java classes to take care of their graphical display. While not providing all the functionality and flexibility of a full meta-mark up language such as XML (Sect. 2), HTML extended with displets could allow all kinds of specialized notations and graphical effects while at the same time leveraging over the existing and well-known set of elements defined by HTML.

Our first experiment with rendering arbitrary, non-text-based mark up extensions [17] was to modify an existing browser to allow the parsing and the visualization of new HTML-like elements. To do so, we took an early version of the HotJava browser, whose source code was freely available, and modified it so that it could accept on-the-fly extensions of the HTML DTD and load the appropriate classes (called *displets*) whenever the newly defined tags were to be displayed. That experiment was extremely limited, in that we used an old version of the Java language, and worked only on a specific version of a specific browser. Furthermore, we heavily relied on the existing rendering architecture of the browser and just provided a minimal effort implementation (basically a displet was just a sequence of drawing instructions for the visualization of the elements).

In [4], on the other hand, we reported about the DispletManager applet, a general, extensible rendering and architecture we have been working on, which can be used for both extensions to HTML and straight XML documents. This architecture is embodied in a Java applet that can be run within any Java-enabled browser such as Netscape Communicator or MS Internet Explorer.

Fundamental design requirements for the rendering engine were:

– it must be possible to create special code for rendering arbitrarily odd data types, in particular non-textual data (*displets*).
– all displets must easily integrate with each other: a chart element may have a mathematical formula as one of the labels, and some staff notation as another, where some notes may act as hypertext links.
– the rendering engine must work both for extended HTML and for straight XML, and the displet classes must be identical.

Figure 1 shows the general structure of the DispletManager applet:

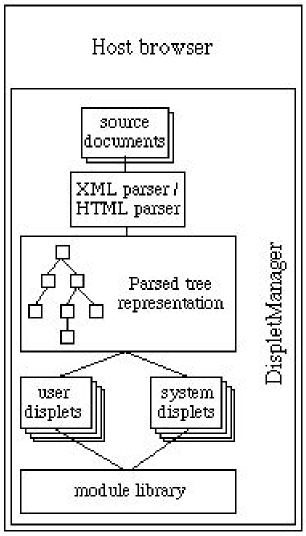

Fig. 1. the general structure of the DispletManager applet

The document chunk to be displayed, be it HTML or XML, is loaded by the displet manager and parsed by the appropriate parser. The resulting tree is then recursively (depth-first) analyzed: the appropriate displet classes are activated to create the rendering (i.e., the display object) of their element on the basis of the rendering of their sub-elements. No class is allowed direct access to the screen: on the contrary, each displet creates a (set of) off-screen bitmap(s) that its ancestor can pass, ignore, modify or add to.

3.1 Extending HTML

Extending HTML is important whenever one needs to provide graphical support for some previously unsupported notation, within a document that can be easily mapped onto plain HTML. The Z notation is one such case, but many others can be devised. The displet manager allows any kind of extension to HTML. In order to extend the HTML tag set, it is necessary to specify the new tags and a bit of syntax constraints, and associate them the corresponding Java classes that will take care of the creation of the visual object.

The following is an example of a simple HTML document with a simple extension for displaying text in reverse video:

```
<applet code="DispletManager.class" align="baseline" width="500"
height="200"><param name="def" value="

<tag name='reverse' hasEndTag
                    nonNesting
                    src='example/reverse.class'>
</tag>

"><param name="HTMLcode" value="

<body>
<p>This is an example of a text rendered in
<reverse>reverse</reverse></p>
</body>

"></applet>
```

The DispletManager is an applet that has two arguments: the first contains the definition of the new tags (in this case, the tag `<reverse>`), while the second one contains the HTML document that has to be displayed, and that includes the extension previously defined.

The standard HTML elements are all pre-defined, and have their own displet classes that are automatically loaded. The 'def' parameter of the applet allows the definition of the new elements, providing some simple syntactical support and the URL of the displet code in charge of providing its rendering.

Upon loading the applet, the displet manager will start the HTML parser and patch it with the elements defined in the 'def' parameter. It will then parse the 'HTMLcode' parameter, verifying that the new elements are being used according to the given grammar. The 'def' parameter acts as a simplified DTD for the new tags: it allows to specify the names, attributes, inclusion rules and minimization features allowed for the new elements, as well as the URL of the Java code containing the rendering applet.

The resulting tree then is examined and the proper displet for each node is activated. Each displet is required to produce a (list of) bitmaps of its content. Each displet may sets parameters and wait for the displet of its sub-elements to return their bitmaps, before producing its own. For instance, the displet for the $< P >$ tag in HTML (that defines a paragraph) sets some values (such as margins, line spacing, font and size) that may affect its sub-elements, waits for all of them to return their bitmaps (one for each word, because $< P >$ combines whole words into lines) and then creates its own list of bitmaps (one for each line).

Figure 2 how the above mentioned document results on screen:

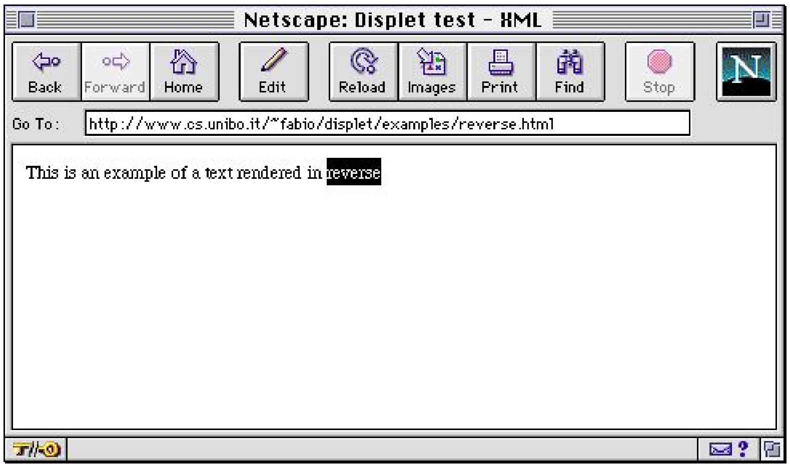

Fig. 2. Rendering a simple displet

We are using a modification and update of a beta version of an old Sun HTML parser. Currently, the HTML displet package contains displets for all HTML 3.2 tags, except tables. Anchors and form elements are included and work correctly. This allows authors to use standard HTML elements within the special elements they define: for instance, it is possible to have styled text, form elements or hypertext links within a Z schema.

4 The rendering engine

The rendering engine used by the `DispletManager` applet consists of a set of Java classes that provide the rendering for the appropriate document elements. These classes are all subclasses of the `DocElement` class, which provides the framework of the rendering procedure.

We are using Java 1.0 because it is supported by most browsers currently in use. We are developing a version for Java 1.1

All classes provide a `createBitmap()` method, whose purpose is to create and return the bitmap of the flow object of the considered mark up element on the basis of the bitmaps of its sub-elements. The `createBitmap()` method is usually not seen by the implementer of new classes, and provides the following functionalities:

- an active drawing environment is managed. The drawing environment is a set of parameters that are used by the rendering methods of the classes in order to decide how to create the bitmaps. For instance, a paragraph-like class may set some parameters that will be used by itself, such as margins, line spacing, alignment, etc., and some that will be used by its sub-elements, such as font name, font size, font color, etc. The `createBitmap()` method allows a displet to set its own attributes with the `setParams()` method, and restores the previous situation when the displet is finished. Since `createBitmap()` methods are recursively activated, this creates a stack that provides the proper parameters at any level of recursion.
- the rendering of sub-elements is managed. The presence/absence of the element in the XSL rule, or some internal decisions for the HTML displet, may cause or prevent the rendering of the sub-elements of the current element.
- the rendering of the element is managed. After the bitmaps of the sub-elements of the element have been created (if appropriate), the `createBitmap()` method calls the `render()` method, which in turn creates the final bitmap (or set of bitmaps) that will be returned. Different classes will implement `render()` differently: for instance, the `render()` method of a block element will collect the bitmaps of its sub-elements in a vertical stack (one above the other), and provide a single bitmap of the whole element, while the `render()` method of a paragraph will collect its sub-elements side-by-side in lines of the given width, and provide a bitmap for every line it has created; this allow the element containing the paragraph to decide how much of the paragraph to display at a time (for instance, in case of scrolling).
- active elements are specified and created. Active elements are those that will need to react to user and system events after they have been displayed. For instance, form elements and anchors have an associated behavior that is activated when the user selects them.

Figure 3 shows the inheritance structure of the classes of the module library:

`DocElements` can either be data, entities or tag elements. DataElement classes are used for the content of mark up elements, i.e., `#PCDATA` in SGML and XML DTDs. They can either be text or hidden elements. EntityElements are provided for the management of XML and HTML entities such as & or the definition of new ones. TagElements are used for the creation of the structure flow objects of the document: they are either flow objects, block objects, inline elements or special elements.

- A block element is a single object that stands alone in the vertical layout of the document. Paragraphs or tables are block elements. A flow element is a block element that is built piecemeal: while plain block elements are built from start to end before the `createBitmap()` returns, flow elements build each of their sub-element and return, and are called as many times as there are sub-elements. This allows long and complex elements to be rendered only for the possibly small section that is actually displayed. For instance, HTML and BODY are considered flow elements, so that the display of an

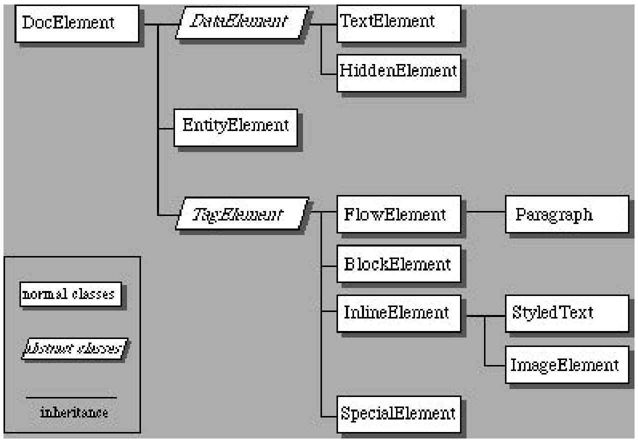

Fig. 3. The inheritance structure of the module library

HTML document can start as soon as the first object is completed, and be interrupted when the available display space is filled.

- Inline elements are elements that can be put side by side with their siblings. Inline elements are used within block elements and may be text-based, images or something else. The StyledText class allows the specification of text runs of arbitrary styles. Inline elements specify the places where they can be broken by creating as many bitmaps as break points. This allows the containing paragraph or block element to determine where the line should be broken.
- Special elements are completely tailorable. While in the previous classes displet programmers can only overload the `setParams()` and `render()` methods, here all methods are overloadable, and can be customized.

As an example, this is the complete source code of the `reverse` displet:

```
package example;
import displet.*;

public class reverse extends StyledText {
        public void setParams(StyledTextParams p) {
                Color c = p.fgColor;
                p.fgColor = p.bgColor ;
                p.bgColor = c ;
        }
}
```

The reverse displet is a subclass of the StyledText, which is a subclass of the InlineElement class. These are classes for text-based objects that behave as in-line elements (eg. bold, italic, etc.). As it can be seen, the programmer of such a displet only has had to specify a parameter and have the `render()` method of its superclass handle all the details. The displets for showing Z specifications are shown in the following section.

5 The Z browser

The main extension to HTML we have considered using displets is the implementation of the complete ZIF DTD. Authors writing Z specifications can create documents containing their Z specifications in a markup language similar to HTML and completely intermixable with plain text and other HTML features such as links, tables, etc.

The ZIF format defines several elements (tags) for the building blocks of the language, such as schemas, definitions, etc., and several entities (literal macros) for the special characters inherited from mathematics and logics. Each element is implemented by a displet that creates a bitmap where the content of the element is appropriately composed and the graphical elements such as boxes, lines, etc. are then added. Entities on the other hand are elements of a graphical alphabet that is contained in a single GIF image and is loaded with the displets.

The following is an example of a Z schema using the Z Interchange Format:

```
<schemadef>
    BirthdayBook
    <decpart>
        <declaration> known: &pset; NAME</declaration>
        <declaration> birthday: NAME &pfun; DATE </declaration>
    </decpart>
    <axpart>
        <predicate>known = &dom; birthday</predicate>
    </axpart>
</schemadef>
```

A schema is defined by a tag called `schemadef`, which contains three elements: the name of the schema, a declaration part and an axiom part. The declaration part contains one or more declarations, and the axiom part contains zero or more predicates. Appropriate ordering and nesting of elements is enforced by the DTD, and is checked when parsing the document. The notations '&pset;', '&pfun;' and '&dom;' are three entities (respectively, the partial set symbol, the partial function symbol and the domain symbol) that will be substituted by the corresponding element in the graphical alphabet containing all the relevant Z symbols. The displet manager can appropriately show document bits as the previous one in a WWW browser.

Since many Z specifiers use LaTeX to produce their Z documents, we have developed an off-line translator called 'Zed2HTML' that transforms Z specifications written in LaTeX using style `oz.tex` into a corresponding HTML document with the appropriate extension.

Of course, the well known LaTeXHTML translator is of little help, as it ignores all LaTeX commands that cannot be immediately transferred onto standard HTML.

For instance, given the following Z specification (the basic birthday book example from [16]):

$$[NAME, DATE]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{__ } BirthdayBook \text{ __________________}} \\
known : \mathbb{P}\, NAME \\
birthday : NAME \pfun DATE \\
\hline
known = \mathrm{dom}\, birthday \\
\hline
\end{array}$$

corresponding to the following LaTeX source document:

```
\documentclass[italian,12pt,twoside,openright]{report}
\usepackage{amsfonts}
\usepackage{oz}

\begin{document}

\begin{zed}
[NAME, DATE]
\end{zed}

\begin{schema}{BirthdayBook}
known: \power NAME\\
birthday: NAME \pfun DATE
\where
known = \dom birthday
\end{schema}
\end{document}
```

The Zed2HTML application transforms the previous LaTeX example in the corresponding extended HTML document:

```
<applet code="DispletManager.class" align="baseline"
width="300" height="700"><param name="def" value="

<tag name='givendef' hasEndTag
          nonNesting
          src='zpack/givendef.class'>
</tag>

<tag name='schemadef' hasEndTag
          nonNesting
```

```
          src ='zpack/schemadef.class'>
    <attr name='id' value=cdata >
    <attr name='group' value=nmtoken >
    <attr name='style' value='vert, horiz' >
    <attr name='purpose' value='state, operation, datatype' >
</tag>

<tag name='decpart' hasEndTag
          nonNesting
          in ='schemadef'
          src ='zpack/decpart.class'>
</tag>

<tag name='axpart' hasEndTag
          nonNesting
          in ='schemadef'
          src ='zpack/axpart.class'>
</tag>

<tag name='declaration' hasEndTag
            nonNesting
            in ='decpart'
            src ='zpack/declaration.class'>
</tag>

<tag name='predicate' hasEndTag
           nonNesting
           in ='axpart'
           src ='zpack/predicate.class'>
    <attr name='label' value=cdata >
</tag>

<entity name='pset' data='#185' font='zpack/zfont.14.gif'>
<entity name='pfun' data='#193' font='zpack/zfont.14.gif'>
<entity name='dom' data='dom' font='TimesRoman%14%Plain'>

"><param name="cod" value="

<body>
<givendef>
    <a name='name'>NAME</a>,
    <a name='date'>DATE</a>
</givendef>
<p>
<schemadef>
    BirthdayBook
    <decpart>
        <declaration> known: &pset; <a href='#name'>NAME</a> </declaration>
        <declaration> birthday:
             <a href='#name'>NAME</a> &pfun;
```

```
            <a href='#date'>DATE</a>
        </declaration>
    </decpart>
    <axpart>
        <predicate>known = &dom; birthday</predicate>
    </axpart>
</schemadef>
</body>
"></applet>
```

The output of Zed2HTML is the HTML specification of the *DispletManager* applet. As in the previous example, two parameters are specified: the definition of the new tags and entities according to the ZIF DTD, and the source document of the actual schema. When run on a WWW browser, the previous documents is shown as in Figure 4.

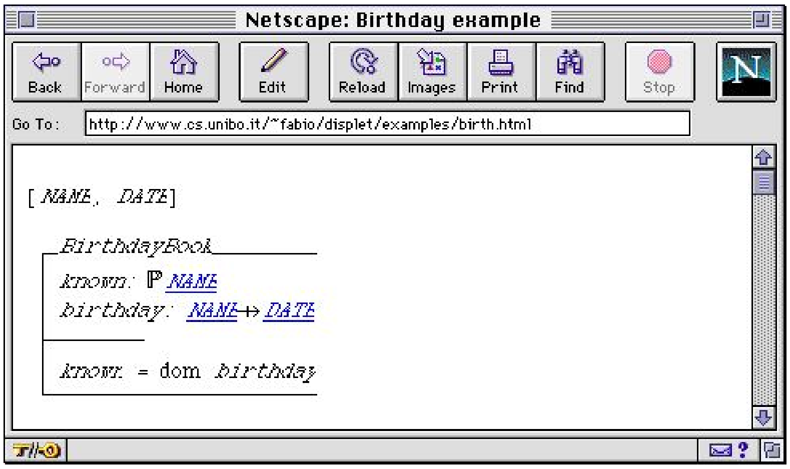

Fig. 4. Visualization on the WWW of a Z schema

A number of issues concerning the HTML document have to be noted:

– In the tag definition part, the new tags are specified with some syntax constraints (for instance, the element `<declaration>` requires the end tag, cannot nest with other declarations, and is only defined within the element `<axpart>`), and the name of the displet class that should take care of its rendering.
– Entities are defined either as textual substitution in specific fonts (such as the &dom; entity, which corresponds to the string 'dom' written using the font TimesRoman 14), or as elements of a graphical alphabet corresponding to a single GIF image. The displet manager will then download the image,

and select and cut the appropriate bitmap from it depending on the position in the alphabet as specified by the entity code in the definition. Thus, in the example, the image of the 'Partial Function' character is the 185th element of the image '`zfont.14.gif`'.
– Z elements and plain HTML elements freely intermix: it is possible to put standard HTML tags within Z schemas, for instance an author may require that some declarations of a schema are written in bold. The Zed2HTML translator automatically connects types used in declarations to their definitions using plain HTML links. The author may freely add or modify the available links and HTML features.

6 Conclusions

We have presented a tool for visualizing Z specifications on the WWW: it fits every browser supporting Java under any platform. The tool is based on HTML extended with 'displets'.

The advantage of having a Z browser that fits all platforms is essentially that sharing of Z documents is encouraged by the diffusion of WWW on the Internet.

A possible application can be a groupware tool for editing and versioning formal documents; such a tool could be integrated with other software tools in order to improve the specification phase of the software process.

The reuse of parts of documents obviously benefits from having these hypertextual Z documents. The tools will also improve the search of pieces of specifications in complex documents: every element in the Z specification can be labeled or linked to other pieces of documents or to a URL on the Internet.

Extended HTML also allows more links for an element (hot word). A possible application of this could be on Z generic schemas: a generic schema specification can provide different links for the parameters of the schema; the links can show the possible schemas or variables that can be substituted for the parameters.

HTML can be further extended in order to include new symbols and integrate Z specification with other notations: new Java classes have to be written for the new symbols.

A displet site is being created at:
`http://www.cs.unibo.it/~fabio/displet.html`.

The site contains the code for the rendering engines, examples for both HTML and XML, and a list of all the displets we have created so far.

Acknowledgments: We would like to acknowledge the help and contribution of Alfredo Rizzi, Stefano Pancaldi, and the help and suggestions of Michael Bieber and Chao-Min Chiu.

References

1. J. Bannan. *Intranet Document Management.* Addison-Wesley, 1997.
2. S. Brien and J. Nicholls. Z Base Standard, November 1992. Programming Research Group.

3. P. Ciancarini, A. Fantini, and D. Rossi. A multi-agent process centered environment integrated with the WWW. In *Proc. 6th IEEE Workshops on Enablings Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 113–120, Boston, June 1997. IEEE Computer Society Press.
4. P. Ciancarini, A. Rizzi, and F. Vitali. An extensible rendering engine for XML and HTML. *Computer Networks and ISDN Systems*, 30(1-7):225–238, 1998.
5. M. Fraser, K. Kumar, and V. Vaishnavi. Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM*, 37(10):74–86, October 1994.
6. D. German and D. Cowan. Experiments with the Z Interchange Format and SGML. In J. Bowen and M. Hinchey, editors, *Proc. 9th Int. Conf. on the Z Formal Specification Notation (ZUM)*, volume 967 of *Lecture Notes in Computer Science*, pages 224–233, Limerick, Ireland, September 1995. Springer-Verlag, Berlin.
7. D. Jordan. CADiZ - Computer Aided Design in Z. In S. Prehn and W. Toetenel, editors, *VDM 91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 685–690. Springer-Verlag, Berlin, October 1991.
8. G. Kaiser, S. Dossick, W. Jiang, and J. Yang. An Architecture for WWW-based Hypercode Environments. In *Proc. 19th Int. Conf. on Software Engineering (ICSE 17)*, pages 3–13, Boston, MA, May 1997.
9. R. Khare and A. Rifkin. XML: A Door to Automated Web Applications. *IEEE Internet Computing*, 1(4):78–87, July/August 1997.
10. L. Lamport. Verification and Specifications of Concurrent Programs. In J. de-Bakker, W. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374. Springer-Verlag, Berlin, 1993.
11. L. Mikusiak, M. Adamy, and T. Seidmann. Publishing Formal Specifications in Z notation on the WWW. In M. Bidoit and M. Dauchet, editors, *Proc. Conf. on Theory and Practice of Sw Development (TAPSOFT 97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 871–874, Lille, France, 1997. Springer-Verlag, Berlin.
12. L. Mikusiak and al. Z Browser: A Tool for Visualization of Z Specifications. In J. Bowen and M. Hinchey, editors, *Proc. 9th Int. Conf. on the Z Formal Specification Notation (ZUM)*, volume 967 of *Lecture Notes in Computer Science*, pages 510–525, Limerick, Ireland, September 1995. Springer-Verlag, Berlin.
13. S. Ressler. *The Art of Electronic Publishing*. Prentice-Hall, 1997.
14. W. Scacchi and J. Noll. Process-Driven Intranets - Life Cycle Support for Process reengineering. *IEEE Internet Computing*, 1(5):42–51, Sept/Oct 1997.
15. C. Sperberg-McQueen and R. Goldstein. HTML to the Max: A Manifesto for Adding SGML Intelligence to the Wold-Wide Web. In *Proc. 2nd Int. WWW Conf.: Mosaic and the Web*, page (Electronic proceedings), 1994.
16. J. Spivey. *The Z Notation. A Reference Manual*. Prentice-Hall, 2 edition, 1992.
17. F. Vitali, C. Chiu, and M. Bieber. Extending HTML in a principled way with *displets*. *Computer Networks and ISDN Systems*, 29(8-13):1115–1128, 1997.
18. Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.