Software Engineering and the Internet: a roadmap

Luca Bompani, Paolo Ciancarini, Fabio Vitali
Dept. of Computer Science,
University of Bologna
Mura A. Zamboni, 7
e-mail: [bompani | ciancarini | vitali]@cs.unibo.it

ABSTRACT
We argue that a roadmap for software engineering and the Internet currently should be based on standards for complex data and document structures, like the Extensible Markup Language (XML). In fact, XML and its cohort of related standards are likely to become an epochal innovation for designing Internet-based software systems. The fields of application of these new notations and related technologies are only limited by human imagination, and simply cannot be enumerated at the moment. Our interests currently concern a concept that we call *declaratively active document*, for which these standards offer an important support. In this paper we describe shortly the state of the art of these new standards, and how we are using the concept of declaratively active document for software engineering purposes.

INTRODUCTION
Internet, and in particular the World Wide Web, have given the software engineering field a distributed environment that is world-sized and actually working. The World Wide Web makes collaboration of remote shops possible, and furthermore it really allows the sharing and collaborative drafting of documents, code and all the objects of the workflow that are part of daily life of the software engineering.

But in our view it is in the field of markup languages that the Internet has provided the most important advancements to the field. Within and outside of the World Wide Web, the Extensible Markup Language (XML, [5]) and its cohort of related standards (XSLT [9], XSL [1], XPointer, XPath and Xlink [12], [10] and [11], DOM [20], RDF [15] and XML-Namespaces [4]) are likely to become an epochal innovation. The fields of application of these new approaches are only limited by human imagination, and simply cannot be enumerated at the moment.

Thanks to XML, any software engineer can define a syntax (i.e., a Document Type Definition, or DTD) tailored for his own needs, and use standard XML tools to create and verify complex data structures which can be exchanged between applications. This makes of XML a very powerful tool for system integration. In fact, the strength of XML lies beyond the capabilities to define community-specific DTDs: for instance, it is becoming convenient to use it for application-specific data objects that are not really meant to be displayed to a human user.

Additionally, XSLT provides much to XML in terms of reach and flexibility. XSLT is a mapping language that can be used to transform an XML document into another one. Its most important use is to transform an XML document into a format that can be displayed by a browser: thus for instance MS Explorer 5 can accept XML documents satisfying an arbitrary DTD and use XSLT to transform them into HTML documents that can then be properly displayed on a computer screen.

XML-Namespace, on the other hand, is a specification to allow the intermixing of tags defined for different document types. Thus, instead of defining a new DTD from scratch whenever a new tag is needed, it is possible to extend it or even to merge different DTDs in a single document. For instance, a mathematician may write the text parts of a textbook using plain HTML and intersperse the HTML with MathML formulas without needing to create a new DTD including the two tag sets.

The extreme flexibility in composition and creation that is granted by the XML language is not yet equated by analogous flexibility in software: currently, browsers only provide a closed and specific set of visualization features (e.g., the HTML + VML [16] object model of Internet Explorer 5), so that any visualization need that exceeds those provided by the tool are naturally frustrated.

In past papers we discussed *displets* ([8] and [3]), our proposal to provide flexible support for special rendering needs that authors of complex documents may have. Displets are software modules (currently they are Java classes) that are associated to each element in an XML document and provide some rendering behavior for that element. Support for the most common element types is given (for instance, text elements), but it is possible at any time to add new modules providing specialized rendering semantics for specific needs.

In this paper we maintain that the architecture previously proposed can be fruitfully used for more than visualization, for it is an extremely general way to associate behaviors to XML elements, and thus to produce active documents that perform computations, enact goals, produce results. The idea behind this assertion is as simple as displets: by associating XML elements to Java classes that are able to perform such activities, XML documents become collections of active components that can be commanded to perform arbitrarily complex computations. This means that displets can be used for both the mere display of information on screen (either via text, or via any specific notation) and the realization of active information elements that provide interactivity, self-update, or any kind of complex computation.

Our approach is particularly useful to make software engineering environments "WWW-aware": the documents of the software process tend to be composed of several different chunks, some of text, some of formulas in special notations, and some of structured graphical diagrams. Currently it is very difficult to turn these documents into pages that can be made available through a Web browser, since each formula and each diagram need to be converted into a passive image. Displets provide a way around these limitations, by allowing the semantically-rich description of formulas and diagrams to be expressed in XML and to be displayed in the browser in their full graphical rendering. Additionally the possibility of activating in some ways the diagrams, verifying their correctness and providing multiple views seem important and easily possible in a general way with the displet approach. Thus some parts of the document can be declared active, rather than executed. We call these documents *declaratively active documents* (or **DADs**) because of this characteristic.

This paper is structured as follows: in section 2 we discuss our idea of active documents, and compare it with similar well-known approaches. In section 3 we discuss the current architecture of displets, and provide examples of some of the displet classes we have created. In section 4 we discuss the concept of active displets and provide some examples, including some applications of both XML and the displet technology to the field of software engineering, and in section 5 we try to derive some conclusive reflections on the application of markup languages and active documents to the future environments of software engineering.

2 ACTIVE DOCUMENTS

Traditionally, electronic documents have been seen as static entities to be subjected to actions, such as displaying or printing, rather than actively taking part to a process. Documents' internal data formats were decided by the applications that created them, which made it very difficult to manage documents as collections of heterogeneous data types, since a different program had to manipulate each one. Furthermore, relations between different documents that are strictly related were difficult if not impossible to express.

On the Internet the old application-centered computation paradigm is slowly fading, and we are moving towards a new data-centered model. Documents do not necessarily belong to a specific application, as in the past, but they may be made of several components that can be independently displayed, printed or made to interact with other components or with the user. It is now possible to build documents with heterogeneous data, coming from different sources and expressed in different formats. This object-oriented paradigm sees thus the document not as the persistent format of some application's data, but as a container of smaller autonomous chunks of heterogeneous data.

Furthermore, the new model makes it possible to build new document types, that were not conceivable previously. Documents can now contain not just static elements such as images or text, but also buttons, fields, and other widget elements that can directly interact with users. Documents become active, they can react to external inputs, can produce computation, can dynamically modify themselves.

2.1 Current generation active documents

Currently, most operating systems provide a generic object model and integrate APIs and services to manage the communication between the objects that compose these complex documents. Many architectures have been defined to support active documents, including OpenDoc (Apple, IBM, etc.), ActiveX (Microsoft), and JavaBeans (Sun).

OpenDoc [14] was an open platform for compound documents defined by CIL (Component Integration Labs, co-founded by Apple, IBM, and other organizations). The ill-fated OpenDoc architecture is based on a few different modules used to organize the content of the documents: the layout system manages the partition of space between document components (called parts), activates the parts and refresh them for the display; the event dispatching system routes events to parts and interacts with the layout system to activate the event target; the storage system helps parts to store their data in one shared compound document. The OpenDoc platform provides a rich set of API that a software developer must use instead of system calls to write parties. This uniformity grants portability of parts across multiple platforms: to move a part to a different OpenDoc platform it is only necessary to recompile it.

ActiveX [17] is Microsoft's platform for active documents, currently available for the Windows operating systems and some UNIX implementations. The name ActiveX does not identify a well-defined architecture, but rather it is the marketing term used to refer to a set of technologies related to the Web and the generation of compound documents. At the core of ActiveX is COM (Common Object Model), a Microsoft standard that specify a way for software components to communicate with each other. It's a binary and network standard that allows any two components to communicate regardless of what machine they are running on, what operating systems the machine are running and what language the components are written in. ActiveX objects thus are generic software components that export their methods and properties through a COM interface. Contrary to OpenDoc parts, ActiveX call directly system call of the operating system they are running on: this allow ActiveX to use all the operating system API, but introduces a lot of portability problems.

JavaBean [19] is the software component model proposed by Sun for Java. A JavaBean is a reusable software component that can be graphically manipulated inside a visual environment. As ActiveX components, JavaBeans can use the full Java API set without any limitation: the JavaBean model define only how a JavaBean can be graphically manipulated but not what it can do, and how it do it. JavaBean model grants a full binary portability, thanks to Java binary portability.

2.2 Active documents with XML

In our opinion, although all these architectures allow documents to become more complex and sophisticated and to include text, structured data, images, multimedia, etc., they suffer from a fundamental weakness: the persistent representation that is used to store the data on disk or to exchange it among applications (i.e., the data format) is usually extremely complex, necessarily in a binary format, and often it explicitly includes the code necessary to create their active components.

Even an HTML document with some Javascript or a Postscript file are in some sense active documents: the former is a displayable document containing some widgets that can exhibit active behaviors, such as type checking or conditional display; the latter is a real program that produces as output the bitmap of the page, and thus can contain any arbitrarily complex computation. Thus in the first example the active part is explicitly contained as executable code, using a different syntax and explicit, cumbersome separators to tell the content of the document from the code; in the second case the active part actually is the document.

We discuss in this paper a new approach to provide generic active documents without including specific executable code in any form, thus differentiating our proposal from all of the above mentioned methods. Our approach relies on XML [5] as a way to express both the content and the active parts of a document, and on executable modules to be loaded dynamically and on demand to provide the activation framework for the active parts.

Every kind of structured information has a chance of being better described and managed using an XML-derived syntax rather than any other traditional one. Examples abound in showing how industry, practitioners, and academics are starting to understand the power and flexibility of a meta-grammar for data formats as embodied in the XML approach. In a sense, XML may become as ubiquitous and obvious for the representations of data structures as ASCII has been in the last thirty years for the representation of Western characters (and XML even considered internationalization issues!)

Many of the DTDs that are being created and will be created in the next future will be thought for structuring XML documents for rendering either on a screen, a piece of paper, or a generated voice. On the other hand, many other DTDs are not even thought for human consumption (CDF and DRP being important examples), but are designed to specify objects and parameters for some kinds of computation. This shows that XML is not necessarily used as a language to structure text documents (or any other specific kind of documents), but, more generally, it is a language to associate some computer-performed activity to data. This activity can be the display of the content for text documents, or the activation of a new channel with some given parameter for CDF, etc.

Of course there are different ways to exploit this innovation. In our opinion, some ways are naturally more elegant and flexible than others, and are more likely to stand longer the test of time. For instance, we believe that writing an XML DTD for a specific data structure, and then creating a closed application around this DTD only exploits little of the flexibility given by XML, and by appropriately using XSL and XML-Namespaces we could deploy particularly sophisticated XML applications.

XSL ([9] and [1]) provides an enormous flexibility in the use of XML, since it allows us to decouple the DTD used by the application from the one used by the human authors: provided a mapping is created between the element names that are meaningful to the users, and the ones that are meaningful to the application, each can work proficiently using their own approach.

XML-Namespaces [4], on the other hand, builds into XML the right to freely mix and match different document types: for instance, when the structure of a piece of data goes beyond the structure as described in its associated DTD, it is possible to re-use elements from another DTD without modifying either one; also, when a piece of data needs to contain two different types of structures, each with its own DTD, it is possible to do so without the need to create a single, combined DTD.

Thus the authoring architecture clearly shows a noteworthy flexibility: the author selects a DTD containing the structuring rules and constructs that best fit the class of documents she intends to write; she then adds the few needed tags that were not present in the original DTD and creates one or more documents exactly matching her own authoring needs. Then via XSL the document is transformed into another one whose elements are known to the rendering application (e.g., a Web browser) and consequently the display is created.

Unfortunately, in the aforementioned architecture, the flexibility stops with the description of documents: the rendering application usually has a fixed set of displayable elements (e.g., text blocks, paragraphs, in-line elements, tables, etc.), and can only show those documents whose rendering needs matches the application's features. In past papers we discussed displets ([8] and [3]), our proposal to provide flexible support for special rendering needs that authors may have. Displets are software modules (currently they are Java classes) that are dynamically associated to each element in an XML document and that provide the rendering behavior for that element. Support for the most common element types is given (for instance, text elements), and it is possible at any time to add new modules providing specialized rendering semantics for specific needs.

In the next section this will be explained in greater detail, and in the following one this idea will be extended to introduce our idea of active document using the technology of displets.

3 XMLC

XMLC (XML Compiler) is our architecture for rendering displets. XMLC relies on technologies and languages such as XML, XSL and DOM, to provide its functionalities.

The main purpose of XMLC is to read an XML document and to produce a displayable tree of Java objects. This happens in a few steps: first, the XML document is read and transformed by a normal XML parser into an internal tree representation based on DOM. Then one or more layers of XSL stylesheets are applied to the DOM tree through the use of an XSLT processor. This creates a final DOM tree where for every element type in the tree there must be an available displet that can be activated. XMLC will finally instantiate all the required displets, creating a tree of runnable objects. Figure 1 shows a schema of the architecture.

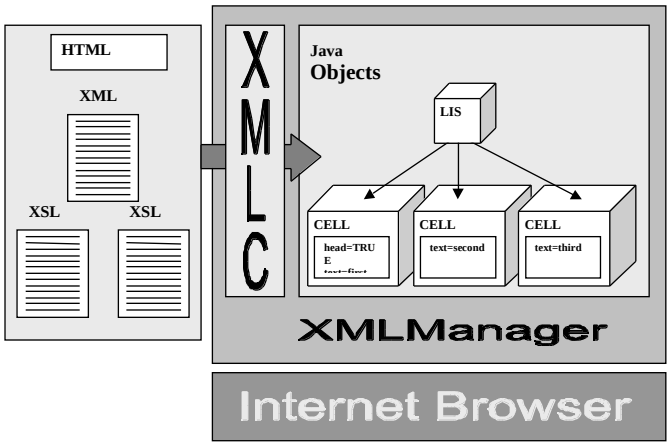

Fig. 1: The architecture of the XMLC application.

Each element in the DOM tree is transformed into a displet according to the following rules:

- the element's name determines the Java class to be loaded;
- the element's attributes determine the settable properties of the instance of the class;
- the element's content (both sub-elements and text content) is added to the tree as children of the class instance.

The current implementation of the XMLC architecture is in Java; a displet can be any sort of Java classes, but using the concept of JavaBeans it is easy to create sophisticated and interoperable displets: the use of JavaBeans Containers and Components, which can be easily organized in hierarchies, nicely fits with the hierarchical nature of DOM trees and XML documents.

Currently, our main use of XMLC is wrapped inside an applet within an HTML document. Parameters of the applet are the XML document to be displayed and the XSL stylesheets to be applied to it. This allows us to display XML documents within well-known Internet browsers, as shown in fig. 2.

Furthermore, since XML elements are transformed into JavaBeans objects, complex behaviors can be easily added during the lifetime of the visualization, providing support for hypertext jumps, animations, interactions with the reader, and in general all the computational capabilities of the Java language.

Fig. 2: An HTML document showing the text displets

In the following we briefly report on the simple, display-oriented displets that have been implemented.

3.1 Text and images

We have implemented support for text oriented XML elements. The level of support is comparable to that of HTML 1.0 text elements: basic blocks (P, UL, OL and header elements) and inline chunks (like I, B, TT elements, etc.), plus an image tag and a simple inline hypertextual

link. In table 1 we show a simple HTML document, and show how this is transformed via XSL into a displayable tree.

```
<HTML>
       <BODY>
       …
       <P>
              This is normal text,

              <BR/>
              this is a new line:


              <B>start bold

                <I>italic</I>
              end bold</B>

              this is normal text

       </P>

       <H1>
              <IMG src="images/java.gif"/>

       </H1>
       …
       </BODY>
 </HTML>
```

```
<Block>
 …
<Paragraph>
  <Word text="This"/><Word text="is"/>
  <Word text="normal"/><Word text="text,"/>
  <NewLine/>
  <Word text="this"/><Word text="is"/>
  <Word text="a"/><Word text="new"/>
  <Word text="line:"/>
  <Word bold="true" text="start"/>
  <Word bold="true" text="bold"/>
  <Word bold="true" italic="true"
text="italic"/>
  <Word bold="true" text="end"/>
  <Word bold="true" text="bold"/>
  <Word text="this"/><Word text="is"/>
  <Word text="normal"/><Word text="text"/>
 </Paragraph>

 <Paragraph font-size="30"
alignment="CENTER">
  <Picture font-size="30" alignment="CENTER"
   src="images/java.gif"/>
 </Paragraph>
 …
</Block>
```

Table 1: A fragment of the HTML document in fig. 2, before and after the XSL transformation

There are three basic Java displets taking care of the display of text elements: Paragraph, Word and MultiWord. A Paragraph is a container spaced vertically (that is, two or more Paragraphs are put one above the other), with parameterized margins, line height and several other aspects. A Word is a component taking care of the display of a single word (separated by variable-width white space). Words are spaced horizontally and can control font, size, style, baseline and a few other parameters of their content. A MultiWord is a container for Words that is still spaced horizontally. It is used to group together Words that share a common propriety (for instance, that belong to the same run of bold characters, or to the same hypertext anchor).

3.2 Hypertext links

W3C is proposing two languages to express hypertext links in XML. XPointer [12] provides a way to express sub-resource addresses within XML documents and other resources, and XLink [11] defines a syntax for hypertextual links between XML documents. XPointers can specify locations within XML documents by collecting progressively detailed location specifiers. This makes it possible to specify an arbitrarily small location without marking it with a tag as in HTML. Instead XLinks extends HTML links by introducing several new features:

- Links can refer to multiple end-points;
- Links can be multi-directional;
- Links can be stored externally to the resources they link;
- Links can be activated in a variety of ways (they may open a new window, substitute the current content, or expand within the current content, etc)
- Links can create groups of related documents to be loaded together.

We have provided a complete implementation of XLink for our XMLC architecture. This has added a few steps to the sequence of transformations of the XMLC application, as shown in fig. 3.

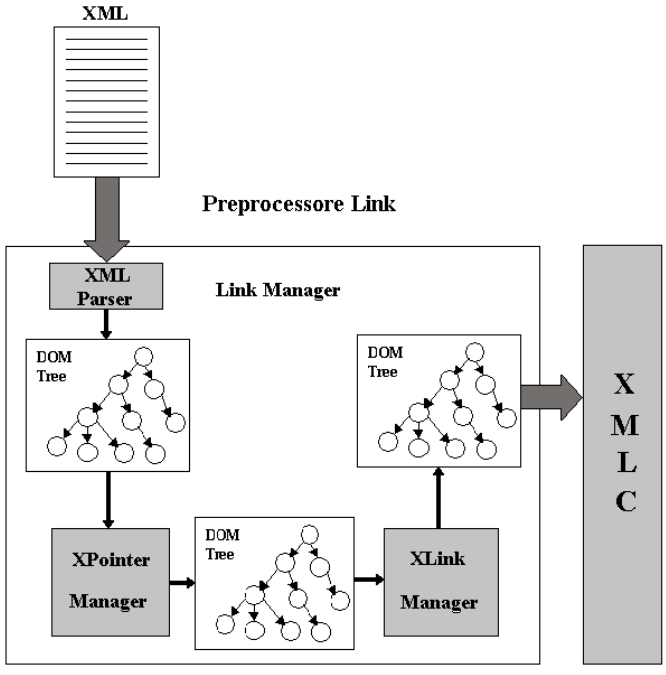

Fig. 3: The XLink-enabled architecture of XMLC

After parsing the XML document, all link elements are identified and added to a list. Then, an identifier is added to all the addressable elements of the document, since after the application of the XSL stylesheets the structure of the document can become arbitrarily different from the original one, and it is necessary to provide a way to identify the elements that can be located through XPointers. The document then are subjected to the usual XSL transformations. Before displaying, though, additional wrapper classes are added around the document elements that are starting points of links, to provide the most appropriate jumping functionality. When the user clicks on one such element, the class reacts, consults the list of destinations, and activates the jump.

The management of document groups we implemented is rather sophisticated and takes into consideration whether the destination document will replace the current one, it will be created in a new window, or it will integrate with the current document. Fig. 4 shows a sample hyperlinked document group.

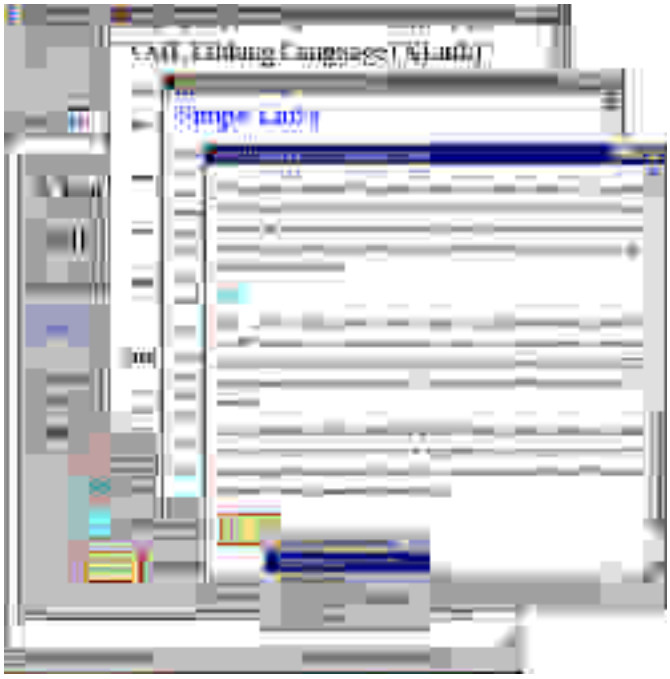

Fig. 4: A simple hyperlinked document group

External links made possible with XLink make it possible to create, for instance, guided tours [2], that is, sets of links that are not stored in the documents, and that are activated on demand to provide additional paths through a set of documents. These documents are not necessarily related in the mind of their authors, but can become so according to the ideas of a third party that may find it important to provide a navigation path through these documents. The external links, therefore, would superimpose to the ones originally present in the documents.

4 ACTIVE DOCUMENTS FOR SW ENGINEERING

Displets provide the flexibility needed to create displayable documents of arbitrary graphic complexity, but they also provide a new way to create active documents: we can in fact associate not just rendering behaviors to XML elements, but any kind of behavior. The “being a paragraph” is a characteristic of an XML elements only when we want to print or display it. When classifying it or indexing it, we may want to associate to it the idea of “being a introductory remark” or of “being searchable”, etc. In general, the application to which the document is subjected provides the semantic framework to evaluate each XML element, and requires each XML element to exhibit a behavior that is appropriate with the purpose of the application itself.

In our framework, then, active documents are XML documents some of whose elements are required to show an active behavior. By providing appropriate stylesheets, XSL can be used to associate a displet to every element of an XML document, according to the needs of the application controlling the stylesheet itself. The displets would then be loaded to perform the appropriate actions and thus behave as required.

It is clear therefore that in this architecture the active documents are not opaque collections of executable objects, or programs or mixed containers of data and scripts, but rather a simple collection of markup and content that are dynamically associated to executable code only on demand, and depending on the application activated. Thus the active part of the document is not a procedural chunk side by side with other declarative parts, but rather it is as declarative as the rest of the passive content of the document, and indeed indistinguishable from them. Active or non active parts only become so according to the stylesheet used, and thus of the application the document is subjected to. This provides an extreme flexibility in associating behaviors to XML elements. We call these documents “declaratively active documents”, because the activity is not procedural, but declared within the document just as the passive content.

The recent advent and present phenomenal success of the World Wide Web, as a hypertext document management system available worldwide to access resources for educational, industrial, and marketing purposes, is strongly influencing also the way in which software processes and related documents are produced and managed.

The XML family of languages provides a solution to most problems involved in writing complex documentation for software processes, that is usually varied in nature, purpose, and contents. Here we shortly recall three important issues: documents involved in a software process are usually structured, are hypertextual in nature, and contain parts of differing type, including some formal notations and requiring special care for the display. Given any software engineering notation, XML can be used to define a uniform abstract syntax useful to integrate different tools specific for such a notation; XML sublanguages like XLink and XPointer can be used to represent hypertext relationships; XSL stylesheets can be used to display software engineering documents inside standard, XML-enabled browsers.

In our research group we have adopted an XML-based approach to build tools for software engineering notations, like for instance Z [6] and Petri Nets. We have developed in the last year a number of specialized browsers/editors for several well known notations, applying systematically the following approach:

Given a formal notation (e.g., Petri Nets diagrams), define a DTD capturing its abstract syntax. This is usually a complex task if the original notation is complex or based on a not well defined syntax. When a DTD is available then it is possible to create XML documents representing the original notation and parse them according to the DTD.

Starting form the DTD, define the XSL stylesheets able to manipulate a document aiming at statically analyzing or transforming it. For instance, given a Petri Net Diagram, a possible static analysis consists of looking for loops. Instead, a transformation could be required to add some behavior in order to render the animation of a Petri Net.

The final step consists usually of enabling the editing and interactive display of the notation inside a Java-enabled browser developing a library of specific displets. We have developed displets for Petri Nets, Z, Statecharts, Data Flow Diagrams, Entity-Relationship diagrams, Workflow Diagrams, and most UML diagrams. Interactive display is possible when some behavioral semantics is associated to the notations. For instance, the Petri Nets displets can play the token game typical of such a notation.

4.1 UML specifications

A key issue is how to define a DTD for a complex sw engineering notation. For instance, if an organization uses the UML family of notations and related development process and tools, it is now available XMI (XML Metadata Interchange, by IBM and others), an XML-based metamodel. All UML documents written according to XMI can be displayed by XML-aware browsers [18] and manipulated by XML-based tools to check for some semantics constraints, like consistency [13]. We are applying our approach to XMI as well. A displet has been developed in our group to provide visualization of XMI. An editor called Elmuth (reverse acronym for *HyperTextual UML Environment*) has been developed. Elmuth is able to create hypertextual and active visualizations of UML documents. Figure 5 shows an instance of MS Explorer including an active document describing (part of) the UML metamodel.

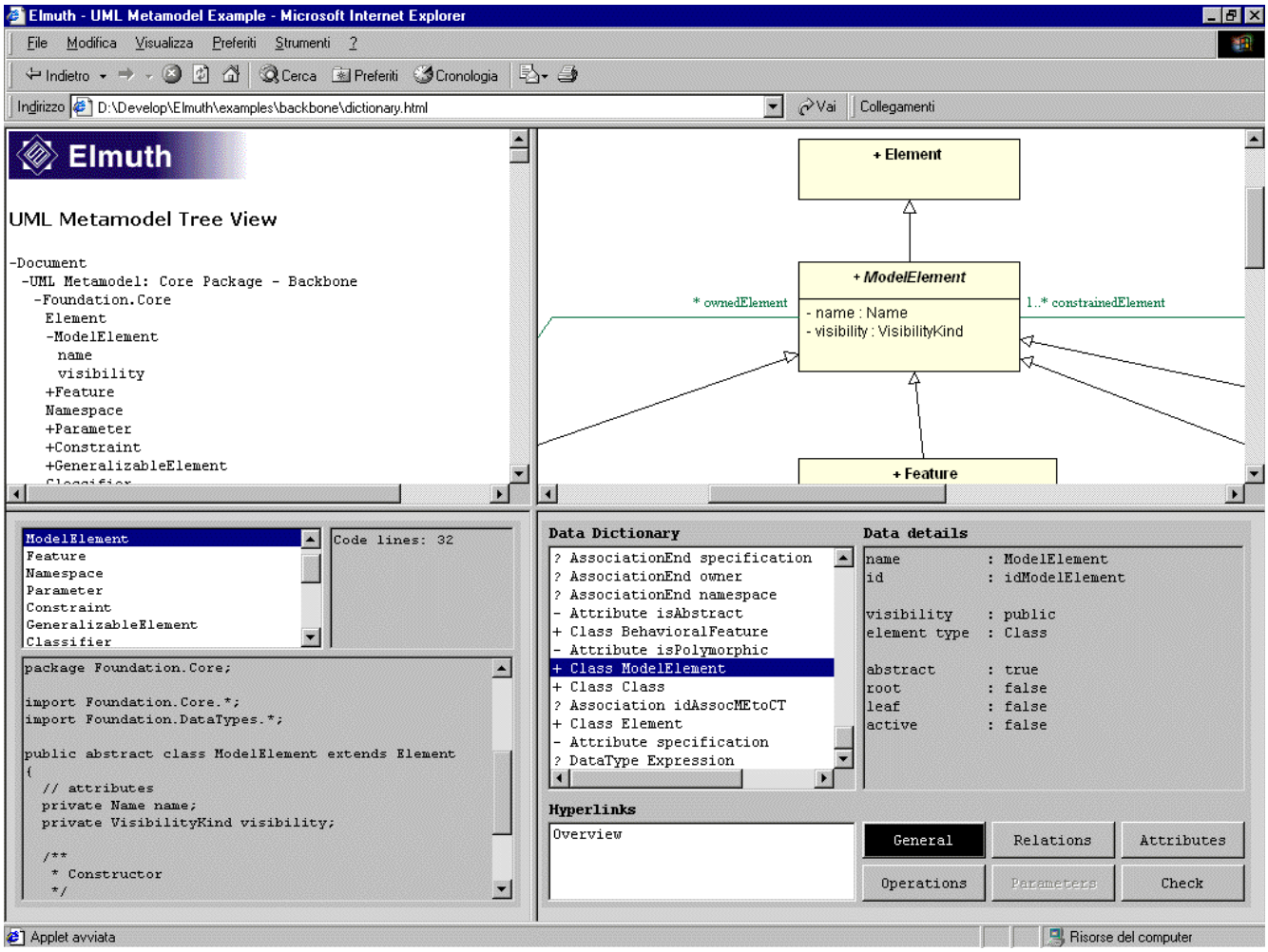

Fig. 5: The representation of a UML diagram

Hypertext multidirectional link among diagrams are managed using our implementation of XLink. The browser includes here four areas: the uppermost left area shows an HTML index useful to navigate the document; the uppermost right area shows a class diagram, the lower right area is a data dictionary, the lower left area shows some code automatically generated from the class diagram.

```
…
<schemadef style="vert" purpose="state">
  PhoneDB
  <decpart>
    <declaration>
      _known: &pset; NAME
    </declaration>
    <declaration>
      phone: NAME &fpfun; PHONE
    </declaration>
  </decpart>
  <formals> K,L,Z </formals>
  <axpart>
    <predicate>
      known = &dom; phone
    </predicate>
  </axpart>
</schemadef>
…
```

Table 2: An example of a Z specification in ZIF

4.2 Other notations

A complete support for the Z notation has been implemented (see also [7]). The DTD for the notation we use is based on the ZIF Interchange Format [6], although, through the use of different XSL stylesheets, other syntaxes can be used as well.

The support for Z elements is provided through the use of a single displet class, zElement, for all the box types that are present in Z specifications (e.g., schema, axioms, etc.), and a special downloadable font for all the mathematical glyphs specific of the Z language (e.g., function, subset, the set of integers, etc.). All other elements of the Z language are mapped onto plain HTML elements such as P, DIV and SPAN. An additional layer of XSL will then transform them into Paragraph and Word objects as needed.

In table 2 we show a small fragment of a Z specification (expressed in ZIF) and in fig. 6 the display of the whole specification in a Web browser.

The development of a notation specific library of displets is not complex. As said, with the help of our students we have developed interactive displets for the most well known software engineering notations, like Petri Nets and Data Flow Diagrams. We have defined special behavioral semantics for such notations, which help in animating the related active documents. For instance, an engineer can play any "token game" interacting with a document including a Petri Net.

Fig. 6: The visualization of the Z specification

5 MARKUP LANGUAGES AND SW ENGINEERING

The World Wide Wed has brought forth many advancements for many fields, including that of software engineering. The advantages of the WWW are clear and well-known: it is the resulting illusion of several easy, simple languages and protocols that work well together and allow developers to compose them to build complex architectures and environments. HTTP, URLs, HTML, server-side computations, client-side scripts have been the foremost elements of the world-wide success of the Web.

Each of the languages and protocols of the WWW are even now evolving to more and more complex potentialities, functionalities, services. The WWW is now a much more complex place than it used to be even a few years ago. This means that it is now possible to build environments of unprecedented sophistication and complexity.

In our opinion, one of the most important recent advancements has been the introduction of the XML family. We strongly believe that the software engineering field, just like many other sophisticated fields, can improve considerably because of the XML family. In particular we single out support for meaningful structure, sophisticated hypertext links and namespaces as the three most important aspects of XML family.

Meaningful structure refers to the possibility to define structures in XML documents and applying to them many sorts of user-defined semantics and composition rules (DTDs now, XML Schema soon). The documentation of a

software project has often to follow predefined structures and should be verified for adherence to predefined composition rules. With XML it is very easy to enforce structure and to provide general means to verify the validity of documents with respect to these rules. Furthermore, specifications expressed with XML syntax can be verified for internal and external consistency partly by expressing their consistencies with DTD rules, that can then be verified with generic XML tools. An important experiment in that direction can be found in [13].

Support for *namespaces* means that it is possible to mix and match different, orthogonally independent document structures. For instance, it becomes possible to describe generally the structure of a document, and then, in certain given special cases, to include chunks of content described by specific rules not contained in the main document class. For instance, it is possible to include Z schemata in a specification document that does not usually allow them.

Finally, *hypertext links* enable software engineers to provide complex functionalities. Links, in XML, are much more sophisticated than with HTML. Of particular importance in our view is that XLink, the linking protocol of XML, supports links that can have multiple destinations, be external and be non directional. The ability to specify non-directional links to multiple destinations makes it possible to express generic relationships between different parts of the documents; external links allow authors to specify different, and possibly independent sets of links on the same documents, for different purposes, readers, or situations.

Applying these functionalities in software engineering would mean providing sophisticated inter-relationship support within the whole software process. Multiple links from each document would point to relevant passages in the documents of all the phases of the process, independent link sets could be activated to enable bird's eye views of the document set, or careful examinations of the specific interrelationships among the parts of a specification, etc.

The one drawback we find in the current state of the XML family is the lack of an adequately flexible activation engine that permits the construction of flexible applications driven by XML documents. These engine could then specialize for screen rendering, or any other kind of relevant computations. We believe that the displet approach is an important step in that direction, providing a generic and flexible environment for any ind of XML-driven computation to take place.

CONCLUSIONS

The XML family is an important step in the direction of a fully fledged document specification language for the Internet. XML and its cohort can actually let users and authors express their data and wishes in a sophisticated, customizable and expandable way. Novel software architectures have to be implemented to take advantage of the generality of these languages. Our XMLC is a very customizable and expandable architecture for displaying XML documents. Being expandable, it has been easy to add support for several sophisticated hypertext functionalities, such as the ones allowed by XLinks and XPointers. Work is under way to add more of them to future implementations.

XMLC is a working prototype, and can be examined, downloaded and used. Elmuth is still under development. See **`http://www.cs.unibo.it/projects/displets/`** for further information.

ACKNOWLEDGMENTS

We would like to acknowledge here the contribution of all the people that have worked on this architecture: Michael Bieber, Chao-Min Chiu, Cecilia Mascolo, Stefano Pancaldi, Alfredo Rizzi, Alessandro Rocca, Alessandro Ronchi, Silvia Villa, and all the students of the 1999 undergraduate course in Software Engineering at the Computer Science Department of the University of Bologna.

REFERENCES

[1] S. Adler, A. Berglund, J. Caruso, S. Deach, A. Milowski, S. Parnell, J. Richman, S. Zilles *Extensible Stylesheet Language (XSL) 1.0*, W3C Working Draft 12 January 2000, http://www.w3.org/TR/2000/WD-xsl-20000112/

[2] M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian, H. Oinas-Kukkonen, 'Fourth Generation Hypertext: Some Missing Links for the World Wide Web', *International Journal of Human-Computer Studies* 47, Academic Press, 1997, p. 31-65.

[3] L. Bompani, P. Ciancarini, F. Vitali, 'Active Documents in XML', *ACM SigWeb Newsletter*, 8 (1), 1999, p. 27-32.

[4] T. Bray, D. Hollander, A. Layman, *Namespaces in XML*, World Wide Web Consortium, 14 January 1999, http://www.w3.org/TR/REC-xml-names

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, *Extensible Markup Language, (XML) 1.0*, W3C Recommendation 10 February 1998, http://www.w3.org/TR/REC-xml

[6] S. Brien and J. Nicholls, *Z Base Standard*, Programming Research Group, Oxford, UK, 1992.

[7] 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.

[8] P. Ciancarini, F. Vitali, C. Mascolo, "Managing complex documents over the WWW: a case study for XML'", *IEEE Transactions on Knowledge and Data Engineering* 11 (4), 1999, p. 629-638.

[9] J. Clark, *XSL Transformation (XSLT) 1.0*, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt/

[10] J. Clark, S. DeRose *XML Path Language (XPath) 1.0*, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath/

[11] S. DeRose, E. Maler, D. Orchard, B. Trafford, *XML Linking Language (XLink)*, W3C Working Draft 20 December 1999, http://www.w3.org/TR/1999/WD-xlink-19991220

[12] S. DeRose, R. Daniel Jr., *XML Pointer Language (XPointer)*, W3C Working Draft, 9 July 1999, http://www.w3.org/TR/WD-xptr

[13] E. Ellmer, W. Emmerich, A. Finkelstein, D. Smolko and A. Zisman. Consistency Management of Distributed Documents using XML and Related Technologies. University College London, UCL-CS Research Note 99/94. Submitted for Publication. 1999.

[14] J. Feiler, A. Meadow, *Essential Opendoc: Cross-Platform Development for Os/2, Macintosh, and Windows Programmers*, Addison Wesley Publishing Company, 1996

[15] O. Lassila, R. R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation 22 February 1999, http://www.w3.org/TR/REC-rdf-syntax

[16] B. Mathews, D. Lee, B. Dister, J. Bowler, H. Cooperstein, A. Jindal, T. Nguyen, P. Wu, T. Sandal, *Vector Markup Language (VML)*, World Wide Web Consortium Note, 13-May-1998, http://www.w3.org/TR/NOTE-VML

[17] Microsoft ActiveX: http://www.microsoft.com/com/

[18] C. Nentwich, W. Emmerich, A. Finkelstein and A. Zisman. BOX: Browsing Objects in XML., University College London, UCL-CS Research Note 99/41. Submitted for Publication.1999.

[19] Sun Microsystems, The JavaBeansTM 1.01, ftp://ftp.javasoft.com/docs/beans/beans.101.pdf

[20] L. Wood, A. Le Hors, V. Apparao, L. Cable, M. Champion, J. Kesselman, P. Le Hégaret, T. Pixley, J. Robie, P. Sharpe, C. Wilson *Document Object Model, (DOM) Level 2 Specification 1.0*, W3C Candidate Recommendation 10 December 1999, http://www.w3.org/TR/DOM-Level-2/