Towards an ”Interaction Sheet” Mechanism in XML Technology

Frédéric Bapst Christine Vanoirbeek

Swiss Federal Institute of Technology
MEDIA group, LITH-DI
Ecublens, CH-1005 Lausanne
frederic.bapst@epfl.ch

Abstract

In the context of XML, this paper develops the concept of a style sheet as an interface between the end user and the encoded information. It argues that the mechanism could be adapted to address the interaction with XML documents, including their modification. An extension of the Cascading Style Sheet language is proposed to express interactive properties. This extension focuses on a set of document-centric interactive manipulations, which could help editing or browsing. Our mechanism, that we called interaction sheets, is especially useful when the application must specify a restricted set of authorized modifications, at a certain step of the document life cycle. When browsers also have editing capabilities, an interaction sheet would be an alternative approach to general XML editors (which are sometimes too permissive) and to Web forms (which are sometimes too restrictive).

Keywords: Structured documents – Internet – XML – Style sheets – HTML – CSS – Editors – Human-computer interaction.

1 Introduction

For a couple of years, the XML technologies have evolved into a major field of interest in the computer science community.

With its ultra-rapid development and propagation, XML already represents both a success story and a milestone in the Internet evolution. Nonetheless, there are still a couple of problems that XML technology is not mature enough to solve elegantly. In this paper, we would like to focus on one family of questions related to the interaction between the human user and XML-formatted information. Various points of view lead to feel the lack of an appropriate mechanism for the expression of some privileged interactive manipulation commands. More specifically, here are the kinds of improvements that we are targeting:

- *style sheets*: both XSL and CSS are good mechanisms to express the presentation of XML documents, but can't we go a step further and extend the style paradigm with some interactive properties?
- *editors*: XML editors are meant to be generic tools, but why not add them a complementary mechanism to help tailoring the user interface to the underlying needs of a particular document in its life cycle?
- *forms*: Web forms organize the user input into a list of named fields, but why not allow more structuring, through directly filling in an XML document, but without losing the current ergonomy supported by forms?
- *browsers*: web browsers have become powerful softwares, but can't we separate the page content from the browsing functionality, and thus help tuning the navigation facilities in a document-centric way?
- *links*: linking being one of the richest feature in document technology, how could a specific document suggest the interactive means that will promote the reinterpretation of its contents as link ends?

At a first glance, it may seem that reaching altogether such a wide variety of improvements is quite a utopy. But in fact we argue that most of the work has already been achieved elsewhere, and that the remaining steps are within immediate reach. What needs to be done is to assemble several pieces of know-how into one relevant mechanism, and to succeed in its wide adoption among the XML community (this latter issue goes beyond our intention).

This paper mainly addresses what we call the *XML input dilemma*: Web forms are meant to input only a fixed list of strings, and general XML editors are meant to input full documents (i.e. any data and structure). We want to support the intermediate situations, when the user is expected to fill in an existing template. For example, there should be a means to express editing constraints at a particular step in the document life-cycle. Our proposition, called *interaction sheets*, is to XML editors what style sheets are to browsers : an interaction sheet specifies how portions of a document can be edited, while a style sheet specifies how portions of a document can be displayed. In fact, we chose to define a unique language that embeds the editing and rendering dimensions, and our proposition is an extension of the CSS language. This was also an opportunity to propose some improvements of CSS not concerned with editing, but with the other questions mentioned above.

The remainder of this paper is organized as follows: Section 2 discusses the requirements of an interaction sheet language, taking five alternative points of views. Section 3 describes our new language, its semantics, and give some usage examples. Finally Section 4 contains some concluding remarks. The whole discussion is also an opportunity to comment the current pieces of XML technology, and to reconsider some of the numerous standards that have recently grown around XML.

2 Requirements for an Interaction Sheet

2.1 Style sheets

The concept of style sheets now seems stable and well-understood. It originally stems from the need to separate the presentation properties from the logical structure of a document. In that sense, CSS is admittedly seen as a major improvement in HTML technology (although current browsers do rarely support the full specification). Existing style sheet mechanisms differ in the richness of the formatting model. For instance, XSL can express more complex structure transformations than CSS. Here we would like to highlight two other strategic features of a style sheet language: the possibility to express instance-specific styles, and the proposition to add editing properties, which is the main idea of this paper.

Generic vs Specific

Our first issue deals with the distinction between specific and generic rules. The underlying philosophy in document engineering suggests that a style sheet describes a generic way of formatting a whole class of documents.

But within a single style sheet some rules are more general than others. For instance, a rule may or not be restricted to elements of a precise type. To avoid interpretation conflicts, both CSS and XSL define a priority measure to determine which rule to apply. At the extreme, both languages may be used to put so many constraints on a rule target, that in fact they allow to designate a single element in a specific document instance. It is even clearly stated with the possibility to attach a rule to the element having a special ID value. When this feature is used, it supposes a conceptual decision which should be emphasized, e.g. by the clear isolation of the specific part. Instead of complaining about the danger of writing specific style sheets, we motivate their benefits, and propose to go a step further.

Let's take an analogy with paper documents. Everyone is used to paint a document with a *pen marker* to highlight some portions, for a later reading. There could be various manners to map this schema onto electronic documents:

a) Get a copy of the document, mark the portions with an attribute to express the highlight status, and let the style sheet encode the desired highlighting property (e.g. background color).

b) Same as (a), but avoid duplication by using a linking mechanism to build the marked version. For instance, XLink can express that the referenced fragment should automatically be displayed at the referencing point.

c) Define an additional style sheet which only modifies the portions that must be highlighted.

Solutions (a) and (b) have the drawback to build an adapted copy of the original document. Even with (b), the copy is not a mere alias, because some parts are modified. Besides, managing lots of links is hard, and the question of which style should apply to borrowed contents is not so trivial. Solution (c) makes more sense because, conceptually, only the presentation has to change and not the document contents, although the visual modification conveys some well-defined semantics. With this solution, the effect of using the marker is not to alter the document, but to alter an associated style sheet (or to build an additional one).

We claim that the marker functionality should definitely pertain to the style sheet mechanism. Consequently, it should be allowed to attach a style not only to a specific element, but also to some portion that do not correspond to a logical element. In XML technology, we could use XPointers to specify the scope of a rule.

From Presentation to Interaction

Our second issue addresses the frontier between presentation and interactive behavior. We claim that there are some rendering properties which may include user interaction. For instance, a collapse/expand functionality can sometimes be a relevant means to enhance the browsing comfort. Current browsers often offer this feature when displaying an XML document which has no style sheet. But it would be even more useful when used in conjunction with a formatted output, provided that the command is accessible only on relevant parts, such as chapters or sections titles.

In fact, a style sheet is intended to make the information more accessible, i.e. to smooth the interface between the user and the information. From this point of view, *looking* is only one kind of human intervention. There is no fundamental reason not to consider the other aspects of interaction. Navigating through the information is a useful sub-task of reading. At the extreme, it may become natural that a style sheet also helps the user to *modify* some parts of the document (cf. Section 2.2), provided that it is the author's intention. The Web shows that documents are not only meant to be seen, but also to interact with.

HTML had the default to mix everything (contents, format, forms, scripts). XML should bring a clear *separation of concerns*. There are some propositions that extend the style sheet language in order to attach scripts to the events (e.g. a mouse click on an element) raised on XML elements (see

e.g. IE5 *behaviors* [10] or the *action language* [2]). It could be argued that specifying an interactive software requires a high expression level, and not only a basic callback mechanism. Typically, a good specification language would address event aggregation or synchronization constraints. It would be interesting to pursue the idea in the field of HTML (scripts, forms, etc.) and XML, in conformance with rich Human Computer Interaction models like MVC [11] or PAC [5]. This might lead to a declarative means lying at the cross of software and document engineering. Related ideas, which could be useful in this field, have already been proposed (cf. the concept of active documents [13], Mozilla XUL proposition [19], UIML [1], or Fuchs's architecture [8]). But our goal here is not to define a general GUI specification means.

Our motivation is quite different: style sheets are centered on documents, not programs. We also believe that the style sheet paradigm should be extended towards interactive properties. But the considered interaction should primarily rest on commands dealing with internal document manipulation. So we won't focus on a general event model (as in HTML scripts or IE5 behaviors), but on a set of general *predefined behaviors* that could be attached declaratively. These behaviors (like modifying text, inserting or deleting elements) are concerned with XML manipulation, not with application-dependent callbacks. The result is a mechanism that offers content manipulation commands on a document, not a language to describe a user interface.

2.2 Editors

In general, an XML document is generated either automatically or through some user input. Here are three typical means to achieve the latter case:

- a general native XML editor;
- a set of Web forms;
- a dedicated user interface, that has to be programmed (e.g. as an applet, using DOM and Java).

With an XML editor, the author has the greatest liberty to compose the document. This is not always an advantage, because the effort of encoding the desired structure is left to the human users. Here are some means to guide them in this task:

- use a highly constrained model, that e.g. specifies mandatory parts; an editor parameterized with a DTD (or some other schema mechanism) is then able to adapt the interaction accordingly;
- start the edition with a document template, i.e. a pre-structured instance with empty contents;
- provide a library of pre-structured fragments, that the user can easily copy, paste, and then modify;
- offer a *macro* functionality, so that the user can record a sequence of commands, and reapply them later somewhere else.

In fact, each of these means has important limitations. A mechanism to express a document-driven assistance in the edition is still missing. A document should be considered within a certain *life cycle*. Sometimes, its modifications follow a planned workflow (or, alternatively, some business workflow contains steps that requires some known modification of that document). In this sense, an interaction sheet would encode the hypotheses that characterize one workflow step, in order to provide the authors with the most relevant set of editing commands.

Beside input restriction, an important feature of editors concerns the front-end philosophy. For many applications, the WYSIWYG paradigm is definitely the best approach, provided that it is supplemented by structure manipulation functionalities. This has been exemplified, for instance by the Thot academic environment [17]. For several years, Thot has been one of the richest editor for structured documents. Basically, the user directly edits the formatted contents. Several views can be activated, and the corresponding editing windows are synchronized. Thot was designed according to some sound concepts [16] in document technology, and uses its own languages to describe document models, presentation rules, structure transformations, etc.

In XML, WYSIWYG editing means that the front-end is based on the result of applying a style sheet to the edited document. This raises two remarks. First, any modification may imply rendering alteration, so the style sheet should be often reevaluated. Second, the more powerful is the style sheet language, the more complex it is to write an editor. That's why writing XML editors based on XSL is quite a challenge, as the mapping from the rendered structure to the original document is not trivial. Consequently, our interaction sheet mechanism comes as a CSS extension. Implementing structure manipulation commands requires at least a strong *selection* management. Then the user interface could provide the operations as contextual menus. We expect commands like element insertion or rearranging, but also a direct access to the values of XML attributes. Attributes are generally not rendered in the formatted view, and editing them should be done in an separate window.

2.3 Forms

The introduction of *HTML forms* was an important step in the evolution of the Web, as it allowed some user input through a standardized way. The data model of the mechanism is a simple set of name/value couples. It is best suited to the input of a data entry composed of several named fields. When the requested information may have a more complex structure (say a hierarchy of nodes), it is necessary to combine several forms, and to write a piece of code to rebuild the desired document at the server side (an alternative is to write scripts, but this is beyond the form concept itself).

On this point, the new XFDL [4] and XFA [7] standards do not bring any real improvement. Bridging the gap means to drop the basic name/value structure and to use instead the structuring power of XML.

XML becomes the universal format for data repository. In the XML technology, the specification of the data can be modeled in a DTD (or another schema language). This DTD can force the presence of some element (or attribute), as a form forces to input a value for a named field. In this context, designing by hand a set of forms that would generate a correct document is a rather wasted effort, because the document model contains many hints. For instance, when an attribute is defined over an enumerated type, the form should contain radio buttons. So a set of input forms can be automatically generated out of the DTD. However, we see two reasons to supplement the DTD with some additional hints:

- *presentation*: the designer certainly wants to visually organize the form elements, or to add some label to describe the different parts; note that CSS2 also solves the latter point, with the `before` property;
- *restriction*: whereas the DTD specifies the complete range of possible documents, one step in the application may need to input only a portion of the document (e.g. when the remaining is automatically generated).

An interaction sheet mechanism would be the declarative language to express these hints.

2.4 Browsers

Current Web browsers are powerful softwares. They are more than a simple interface to the HTTP protocol. Here is a typical set of additional features: formatting engines (CSS, XSL), interpreters for script languages, various Internet services (mail, news, ftp), XML parser, security mechanism, GUI model (HTML forms and events), etc. Browsers are often distributed with their own HTML editor. By the way, this shows that the technology is indeed mature enough to combine browsing and editing, as suggested in this paper and simulated in the Amaya [14] project. But staying at the browsing level, the current technology does not allow to express the navigation capability as a parameter of the document.

Basically, *browsing* means (i) to visualize the information in a convenient way, and (ii) to navigate in the information, be it internal or external to the current document. Visualization is well supported through style sheets. As to the navigation, we distinguish three typical means:

a) to scroll the page;

b) to follow an explicit link;

c) to reach one of the logical element of the current document.

Let's focus one (c), which suggests the notion of *table of contents*. In the HTML world, it is implemented typically using an additional window (or frame) giving access to the main entries of the document.

With the current technology, to publish an XML document on the Web and offer a table of contents navigation help, it is necessary to write a complicated XSL style sheet, which would generate HTML frames or scripts. We believe that the effort is quite high for such a basic functionality. The same argument applies to the collapse/expand (cf. Section 2.1) functionality. With XML, the Web is intended to access the information in its original form. For the authors, HTML seems a too low level model to encode the navigation support. This is yet an important task, that the machine cannot automatically guess: only some element types are useful for an outline view.

It is right that browsers will soon enhance the browsing power when they correctly handle XLinks. But this only concerns *explicit* links, i.e. relations between components that the author decides to specify. We expect an interaction sheet mechanism to address the specification of *implicit* links, i.e. links that are not part of the intrinsic information, but that are useful to browse it. Current Web pages often turn implicit links into explicit ones, e.g. when they contain 'back-to-top' anchors.

So we claim that the browsers should offer a standardized means (as is the 'history' or 'bookmarks' menus) to support the navigation among implicit links. Our interaction sheet mechanism makes it easy to associate one or several *index sets* (e.g. list of figures) to an XML document.

In fact, the table of contents feature would be elegantly solved, provided that XML browsers all support *synchronized views*. When several style sheets are available for an XML instance, the user could open each one in a different window. To synchronize these views means to adapt every scrolled region to any user selection command (e.g. a click on an element). There is just a difficulty (already mentioned in Section 2.2 about WYSIWYG editing), namely the non-trivial matching between logical elements and formatting flow objects.

2.5 Links

In the document technology, links play a key role: they help both to avoid information duplication, and to express structures different from hierarchical patterns. That's why XML is supplemented with two languages (XLink and XPointer) dedicated to linking. Once encoded, an XLink does convey rich information. But format standardization is only one issue related to linking. Links *exploitation* and links *generation* tools are two others. The former is a currently hot topic (e.g. parser support, browser behavior). We claim that the latter also deserves some attention, because rich links are harder to edit.

HTML links are simple enough to allow an easy input. Not only the link model is restricted to single (two ends), inline

(one end being the link expression) relations, but the pointing mechanism is very limited. Indeed, the URL either describes a full document, or an identified fragment. In both cases, the URL has rarely to be typed in, because it is easy to copy it from a browser (either the location or its associated targets). The only choices concern the relative or absolute nature of the URL.

With a richer pointing mechanism like XPointer, the question is less trivial. There is a huge number of potential targets (not just the nodes) in a single document, and each target can be referenced by an infinity of correct XPointers. The issue is important, because there is no systematic way to find the most relevant XPointer for a target. In case of a bad choice, the risk is to break the link, e.g. when the fragment is moved. Depending on the application, such modifications may be frequent. The right solution depends on the data context, and may not always follow some naive heuristics (like stepping down from the last node owning an ID).

Of course, the best way to anticipate a link end is to put an ID to the element. But it is not always possible: the interesting portion may not correspond to an element, or the web publishing service has not the right to modify the document.

When an application requires the manual input of links, any support constraining the XPointers has to be specifically programmed. Our interaction sheet mechanism offers to express such constraints declaratively, so that the editor can itself assist the user in his linking task.

It is worth noting that besides links editing, our proposition is also useful for mere browsing, e.g. when the user maintains a list of bookmarks at a sub-document level.

Finally, it could be argued that the desired effect (providing a relevant XPointer to a certain target) could be obtained in a separate document, containing a set of links. But the primary goal of links is to express logical relations among document parts, not to suggest how a portion should be referenced.

3 Sketching a New Mechanism

3.1 Basic Principles

Basically, we chose to reuse the general philosophy of the CSS standard, which proves to be powerful and easy to understand. An interaction sheet contains a list of rules, each associating a property value to a *selector*, i.e. the scope of the rule. The mechanism is specified through (i) the list of properties with their possible values, and (ii) a combination protocol based on a specificity measure.

In fact, we decided to stay even closer to CSS. Because presentation is part of interaction, any CSS style sheet should be a valid interaction sheet, without any change in the semantic behavior.

Our extension of the current CSS standard is twofold. First we extend the selector language. Second and most important, we define new properties and their underlying semantics.

Note that we use the original CSS syntax. It could be redefined in an XML format, to take advantage of existing tools like parsers. This would not influence the concepts itself, only the implementation.

3.2 Selectors

As already stated in Section 2.1, we propose to allow any XPointer [12] as a rule selector. This way, the properties may also be attached to a portion that does not correspond to a document node. The argument is that the information structure is not always the right one for a particular task like rendering. This was already admitted in CSS2 with the notion of pseudo-elements. Offering the whole power of XPointers may be very general, but it takes advantage of an existing standard, and opens the field for unexpected uses.

There should be a syntactic hint to determine the presence of an XPointer instead of a CSS-like selector. For instance, an XPointer selector may be surrounded by something like `xptr(...)`.

Note that the use of XPointers as a selection mechanism is orthogonal to the use of properties. The main goal of interaction sheets, namely the support of constrained editing, does not at all require a change in the selector language. But as we define an extension to CSS, we take the opportunity to suggest a refinement of the selector means.

3.3 Interactive Properties

In this section, we propose a set of new properties that meet our requirements. The major choice concerns the management of the allowed editing commands. A simple Read/Write flag (in CSS, 'display:none' could yet be interpreted as "unreadable") is not sufficient, because we want to provide a finer control over the document modifications. Our proposition consists of two properties (`editable` and `editable-attr`), and makes it possible to update only one attribute value, or to reorder sibling elements without the right to alter the text. Here we suppose that the selection is always allowed (and can be used in a 'copy' command), but the modification (e.g. a 'cut' or 'paste') must be accepted by the interaction sheet.

We don't want to restrict the possible values of the input data: this should be done at the model level (DTD or other schema language), and we don't see a good reason to impose any further constraints. For instance, when the DTD defines an attribute as type IDREFS, we expect the browser/editor to reflect this constraint, typically via a multiple selection list giving all existing IDs.

The three other properties (`collapsible`, `anchoring` and `indexed-in`) are minor improvements suggested in Section 2.

Property `editable`**.** This property describes the editing rights allowed on the target. The following values are defined (and can be combined in a list when it makes sense):

- *none*: no modification is allowed (default);
- *text*: it is allowed to modify the textual content at each point of the target;
- *insert*: it is allowed to insert new elements everywhere in the target (according to the DTD if available);
- *reorder*: it is allowed to move any (completely contained) sub-element within the target (according to the DTD if available);
- *full*: it is allowed any modifications in the target, such as editing the text, add or remove sub-elements.

Property `editable-attr`**.** This property must be used only when the scope is at the element level (not on sub-element XPointers). It indicates which attributes are subject to modification. The following values are defined:

- *none*: no attribute can be edited (default);
- *all*: all attributes can be edited (according to the DTD, when available);
- *only(att1 att2 ...)*: only the given list of attributes can be edited;
- *all-but(att1 att2 ...)*: all attributes can be edited, except those given in the list.

Property `collapsible`**.** This property describes how the target should behave as to the collapse/expand functionality. The choice of how to provide the commands is left to the browser. On visual agents, we typically expect either a small button at the beginning of the target, or a contextual menu. Browsers may offer global commands like 'collapse-all' or 'expand-all'. The following values are possible:

- *none*: the target is not subject to the collapse feature (default);
- *on*: the target supports collapse/expand, and is initially collapsed;
- *off*: the target supports collapse/expand, and is initially expanded.

Property `anchoring`**.** This property indicates one (or several) XPointer that is intended to serve as a relevant expression for referencing the target. Most often, this XPointer will directly point at that target, though it is not mandatory. The browser is free to decide how to use this information. On visual agents, we typically expect a contextual menu (available when the selection is within the target) with the effect to copy the XPointer into the clipboard (for later pasting as an attribute value in any link element). The `anchoring` property is relevant in situations when the target is likely to be reused as a link end (but has no ID), and the direct path from the root element is not the most robust way to reference it (e.g. some rearrangements are expected).

Property `indexed-in`**.** This property is used to add the target to one (or several) *index set*. An index set is intended to provide a collection of implicit links that may be useful for browsing. An index set is referenced with an id (as are counters in CSS), that the browser may use as the title of a related menu. As described here, the `indexed-in` feature does not give much control on the organization of those index sets, as it gives a mere list. It is hard for the browser to guess a relevant indented display that would reflect the structure of the outline (e.g. sub-sections). As we already stated (cf. Section 2.4), this property will be of little use when editors/browsers support synchronized views.

3.4 Execution Model

Rule Interpretation. The previous section explained the meaning of our proposition about new properties. There is yet to define how a list of rules should be interpreted. The selector language extension requires an adaptation of the scope computation. In fact, we must extend the CSS specificity measure to allow for XPointer selectors. Here is an informal possible definition. An XPointer selector is more specific than any CSS selector. If an XPointer references a target that is entirely contained in the target of another XPointer selector, then the former is more specific. In the remaining cases, the rules lexical order determines the specificity.

Update Protocol. Adding editing commands to CSS raises the question of storing the updated document. Basically, we suppose that the editing commands only affects the local copy of the document, at the client-side. This confirms a general shift of capabilities from servers to clients in Web design. But most applications will need a way to transmit the modified document back to the server. The basic HTTP protocol (e.g. with the PUT method) is probably not powerful enough, if we consider concurrent accesses and the lost-update problem. So our interaction sheet mechanism would better be used in conjunction with a platform like WebDAV [18], which allows for concurrency and versioning.

User Interface. Our proposition supposes that several commands are available when browsing a document through its interaction sheet: collapse/expand, type text, insert/move/delete elements, edit attributes, get an XPointer expression for the current selection, etc. We do not want to impose the detailed user interface, because this would put strong assumptions on the browsers philosophy, or even

on the media type. But we clearly have in mind a kind of WYSIWYG behavior.

Dynamism. The extension of CSS towards document editing commands raises an interesting problem about dynamism: which rule should apply to the parts being modified? Presently, we suppose that the end user has a total control over the document portions that he is editing. In other words, the interaction properties should not be re-evaluated after each user intervention. Otherwise, there is a risk that the user gets or looses access rights in an ill-timed way. But we believe that the question deserves a deeper analysis. By the way, the CSS specification also defers that question of dynamism (e.g. when a script modifies the dimensions of an element).

3.5 Example: Requests for Proposals

In this section, we discuss a business situation where document management is of central importance. The case study concerns requests for proposals. In fact, it is this application that made us feel the lack of an interaction sheet mechanism. The examples below illustrate the use of our interactive properties. Only standard CSS selectors are used.

3.5.1 Overview of the Process.

Requests for proposals (RFP) represent an important business-to-business process. This is a kind of demand-driven commerce. Roughly, the buyer publishes the project specification as an RFP, then the potential suppliers submit their offers, and finally the buyer chooses the best candidate. Inbetween, there is usually the possibility to discuss ambiguities in a forum-like way. An emitted RFP contributes to (i) specify the buyer's needs, (ii) to officially and widely announce the buying intention, and (iii) to prepare the comparison step by constraining the contents of the proposals.

There is now an increasing interest to design electronic platforms acting as RFP market places [3]. In this context of electronic commerce, document formats like HTML/XML play a decisive role. Several other factors make RFPs interesting with respect to an interaction sheet mechanism: (i) the documents are shared between team members, both for reading and writing; (ii) the structure of the offers is part of the requirements; (iii) the evaluation report contains many rating, comments, and comparison links; (iv) the documents are mainly designed *by* and *for* human brains.

3.5.2 Editing a Request for Proposals.

An RFP is a strategic document with which several business services are concerned (marketing, production, finance, etc.). Thus it is often the result of many contributors, under the responsibility of a project leader. The leader may have the exclusivity of the document editing (through the agglomeration of all other pieces), but the process could also be driven in a more document-centric way, with shared accesses. With a set of interaction sheets, it would be possible to assign each part of the RFP to the right employee or service.

For example, the commercial service may be responsible to compose only one section of the document. Its interaction sheet could then contain the following rules:

```
section: {   editable:none
          collapsible: off;
           indexed-in: paperOutline;}
section[id="sct.comm"]:
            {editable: full;
          collapsible: on; }
```

The DTD could define an attribute `isLegal` intended for the legal service to approve the content of each section. This service could receive the RFP with an interaction sheet containing:

```
section: {editable-attr:only(isLegal);}
```

3.5.3 Editing a Proposal.

In the RFP process, the buyer prepares the canvas of the offers, in order to anticipate the comparison phase. There is often one part of the RFP which is a ready-to-fill offer template. Its structure is imposed by the buyer, and the contents is under control of the supplier. The RFP document could be supplemented with an appropriate interaction sheet reflecting this schema. Let's suppose that the section divisions have an attribute `isLeaf` saying whether or not the template imposes subdivisions. Then here is a possible excerpt for the interaction sheet provided to the suppliers:

```
part[id="offerTempl"] section[isLeaf="yes"]:
            {editable: full;}
part[id="offerTempl"] section[isLeaf="no"]:
            {editable: text;}
```

Using such a sheet would insure the supplier that his offer keeps conform to the buyer's intentions. Note that writing an offer goes far beyond filling in a form, and requires a real editor. For instance, the supplier may consider that a requested sub-section needs a further decomposition. Nevertheless, the buyer wants to ensure that the corresponding document respects his template (section names, perhaps even attributes that will be used by the evaluation framework). This is a typical case where *highly constrained editing* is needed, showing the main advantage of our interaction sheet mechanism.

3.5.4 Evaluating the Proposals.

The evaluation phase is reported in a document, called the evaluation report, for which there are usually several writers. But distributing the access rights among the evaluation

team is not the only possible application of our mechanism. For instance, the buyer maintains a collection of criteria, that are to be used for the evaluation. Maybe these criteria are known to be related to precise parts of the offer template. In order to guide the reading of the proposals, we could provide one (or several) index of criteria, to point to the corresponding portions of the common offer template. So every offer could be displayed with the same interaction sheet holding the most useful locations, e.g.:

```
part[id="offerTempl"] supplier nbEmployees,
part[id="offerTempl"] supplier location,
part[id="offerTempl"] section[id="warranty"]:
         {indexed-in: admissionConditions;}

part[id="offerTempl"] section[id="experience"],
part[id="offerTempl"] section[id="pricing"],
part[id="offerTempl"] section[id="delays"]:
         {indexed-in: basicCriteria;}

part[id="offerTempl"] section[id="summary"],
part[id="offerTempl"] section[id="remarks"]:
         {indexed-in: overview;}
```

Finally, in the evaluation report, most of the material (comments, ratings, etc.) will contain references to portions of the offers, so links management is important. However, we don't see an obvious application of the `anchoring` property in that particular example; one reason is that the offer documents are not supposed to change during the evaluation, and thus the robustness of link ends is not threatened.

4 Conclusion

In this paper, we have proposed an interaction sheet mechanism as an additional stone in the XML technology. This work is driven by two major motivations:

- to reach a better separation of concerns between information and its access, i.e. between an XML document and the way it is submitted to the user intentions;
- to solve the XML input dilemma (editors are too permissive and forms too rigid), and let the editor/browser guide the user thank to a specification of the authorized document modifications.

We follow a general trend in Web technology, namely expressing the most useful customization tools in a declarative way. The XML machinery should offer simple means to solve common needs, whereas any non-standard applications can always rely on dedicated software development. Our proposition agglomerates different ideas. Some of them represent only shallow changes, like adding to CSS the commands for collapse/expand and table of contents. But we also bring a few deeper innovations, like the use of XPointer for rule scoping, and above all the support for document-centric editing. There are also some ideas that are probably not mature enough, such as the `anchoring` property.

In general, XML is a good opportunity to go towards an environment aware of the document life cycle. This paper is an attempt to open the discussion and to encourage some implementations. As to the perspectives, we expect two direct extensions of the present proposition:

- Integrate our mechanism with the access rights aspects of the *collaborative edition* approach (cf. Byzance [6]). We could forbid at the server side the access to the document without the authorized interaction sheet. It may be useful there to consider encryption techniques, if the interaction sheet serves as an authentication means.
- Define a mechanism to express the whole life cycle of a document. This means declaratively connect together several interaction sheets, thus linking each step of the planned document evolution.

Combined together, these two improvements would lead to an environment that fully specifies the *workflow* associated with the document manipulation, from the user interaction perspective. This is certainly a promising objective to draw document management and business workflow (for which documents are used) closer together.

Another issue will deal with the interactive specification of an interaction sheet. What user interface commands would be appropriate when editing an interaction sheet? Note that this is already an interesting topic in CSS engineering [15], and a promising approach is model inference from a set of formatted samples.

Finally, the crossroads of XML and Human Computer Interaction could be the source of completely different research directions. One would deal with multimedia documents (i.e. with time dimension), which often imply a kind of interaction. Section 2.1 suggested the question of specifying in XML a detailed user interface. As another issue, it would be interesting to design a general framework to trace the user interventions as an XML document: given an interactive software and the list of possible commands, the goal would be to report the sequence of interaction events occurred during a session. This would be a concrete step towards the manipulation of an *interaction algebra*, as defined in some Human Computer Interaction models like the User Action Notation [9].

References

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. In *Eighth nternational Wrld Wide Web Cnference(WWW8)*, Toronto Canada, May 1999. `http://www.harmonia.com/papers/www8/`.

[2] Vidur Apparao, Brendan Eich, Ramanathan Guha, and Nisheeth Ranjan. Action Sheets: A Modular Way of Defining Behavior for XML and

HTML . Technical report, W3C, June 1998. http://www.w3.org/TR/NOTE-AS.

[3] Frédéric Bapst and Christine Vanoirbeek. XML Documents Production For An Electronic Platform of Requests For Proposals. In *International Workshop on Electronic Commerce (WELCOM'99), part of the 18th Symposium on Reliable Distributed Systems (SRDS'99)*, Swiss Federal Institute of Technology, Lausanne, October 1999. IEEE.

[4] John Boyer, Tim Bray, and Maureen Gordon. Extensible Forms Description Language (XFDL) 4.0. Technical report, W3C, September 1998. http://www.w3.org/TR/NOTE-XFDL.

[5] Joëlle Coutaz, Laurence Nigay, and Daniel Salber. Agent-Based Architecture Modelling for Interactive Systems. In David Benyon and Philippe Palanque, editors, *Critical Issues in User Interface Systems Engineering*, chapter 11, pages 191–210. Springer, 1996.

[6] D. Decouchant, M. Romero-Salcedo, and M. Serrano. Principes de conception d'une application d'édition coopérative de documents sur internet. In *IHM'97*, 1997.

[7] Gavin F. McKenzie et al. XFA-Template and XFA-FormCalc. Technical report, W3C, June 1999. http://www.w3.org/1999/05/XFA/xfa-template.

[8] Matthew Fuchs. The User Interface as Document: SGML and Distributed Applications. In *Computer Standards and Interfaces*, volume 18, 1996. http://cs.nyu.edu/phd_students/fuchs/index.html.

[9] Deborah Hix and H. Rex Hartson. *Developing User Interfaces – Ensuring Usability Through Product & Process*. Wiley, 1993.

[10] Alex Homer. *XML IE5 Programmer's Reference*. Wrox, 1999.

[11] P. Johnson. *Human Computer Interaction – Psychology, Task Analysis and Software Engineering*. McGraw-Hill, 1992.

[12] Eve Maler and Steve DeRose. XML Pointer Language (XPointer). Technical report, W3C, 1998. http://www.w3.org/TR/WD-xptr.

[13] V. Quint, I. Vatton, and J. Paoli. *User Interfaces for Symbolic Computations*, chapter Active Structured Documents as User Interfaces. Springer Verlag, 1996.

[14] Vincent Quint and Irène Vatton. Using Amaya. Technical report, W3C, 1999. http://www.w3c.org/Amaya.

[15] Hélène Richy. Document Style Design by Direct Manipulation. In Roger D. Hersch, Jacques Andr, and Heather Brown, editors, *Electronic Publishing, Artistic Imaging and Digital Typography (RIDT'98)*, number 1375 in Lecture notes in computer science, March 1998.

[16] Cécile Roisin and Irène Vatton. Merging Logical and Physical Structures in Documents. In *International Conference on Electronic Publishing, Document Manipulation and Typography (EP'94)*, Electronic Publishing, pages 327–337, April 1994.

[17] Irène Vatton, Cécile Roisin, and Vincent Quint. Thot Reference Manual. Technical report, INRIA, 1998. http://www.inrialpes.fr/opera.

[18] Jim Whitehead and Meredith Wiggins. WEBDAV: IETF Standard for Collaborative Authoring on the Web. *IEEE Internet Computing*, pages 34–40, September/October 1998. http://www.webdav.org/papers.

[19] Dave Yatt. Introduction to a XUL Document. Technical report, Mozilla Org., 1999. http://www.mozilla.org/xpfe/xptoolkit/xulintro.html.