

VXT: A Visual Approach to XML Transformations

Emmanuel Pietriga
Xerox Research Centre Europe
6 chemin de Maupertuis
38240 Meylan - France
pietriga@xrce.xerox.com

Jean-Yves Vion-Dury
Xerox Research Centre Europe
6 chemin de Maupertuis
38240 Meylan - France
vion-dury@xrce.xerox.com

Vincent Quint
INRIA Rhône-Alpes
655 Avenue de l'Europe
38334 Saint Ismier - France
quint@inrialpes.fr

ABSTRACT

The domain of XML transformations is becoming more and more important as a result of the increasing number of applications adopting XML as their format for data exchange or representation. Most of the existing solutions for expressing XML transformations are textual languages, such as XSLT or DOM combined with a general-purpose programming language. Several tools build on top of these languages, providing a graphical environment. Transformations are however still specified in a textual way using the underlying language (often XSLT), thus requiring the user to learn the associated textual language.

We believe that visual programming techniques are well-suited to representing XML structures and make the specification of transformations simpler. We present a visual programming language for the specification of XML transformations in an interactive environment, based on a zoomable user interface toolkit. Transformations can be run from the application or exported to two target languages: XSLT and Circus, a general-purpose structure transformation language designed by the second author and briefly introduced in this paper.

Keywords

Visual programming languages, XML transformations, XSLT, Circus, Zoomable User Interfaces

1. INTRODUCTION

As XML [6] is now a very popular standard, more and more applications adopt it to represent, store, and exchange documents and data. But XML is a meta-language and its wide adoption leads to a number of application- or industry-specific languages. All those languages share the same structuring principles and the same syntax, but they are still different. On the other hand, documents encoded for a specific application, using the XML language of this particular application, often have to be processed by another application, which uses a different XML language. Transformations are then needed, and given the wide variety of XML languages, transformations play a key role for XML documents and data.

A number of languages are employed to specify XML document transformations. Up to now, all those languages have a textual syntax, and require some programming skills from their users. In this paper, we take a different approach and present a visual language and its environment, to make document transformations easier and to allow more users to take advantage of XML document transformations.

1.1 XML document transformations

Document transformations cover a broad range of applications. The most frequent transformations on the Web today are certainly those related to rendering. As XML allows us to represent a document independently of its presentation, there is a strong need for turning a structured document into a representation that is better suited for presentation. That is the goal of the XSL language [15], which includes an XML vocabulary for specifying formatting semantics (XSL Formatting Objects) as well as a language (XSLT) for transforming XML documents. Although XSLT was initially designed to describe how an XML document is transformed into a structure of XSL Formatting Objects, it is actually powerful enough to specify other kinds of transformations. XSLT is now often used to transform XML data and documents into XHTML pages, that can then be viewed with standard Web browsers. SVG [16] is another format, also based on XML, that can be used to render XML documents.

But formatting is not the only purpose of document transformations. Organisations use many different formats for representing their documents and data, and they need to transform from one format to another when it comes to data exchange. Even within the same organisation, it may be necessary to adapt a document in order to use it in a different context. Re-purposing documents and data is thus another major application of document transformations.

Finally the use of XML in databases and the emergence of query languages suited to XML structures such as XQuery [10], tend to blur the frontier between documents and data, making more data sources available to the XML world, and calling for even more transformations.

1.2 Languages for XML transformations

We have already mentioned XSLT as a language for specifying XML document transformations, but there are clearly alternatives. Using both a general-purpose programming language and the DOM API [19] provides the greatest flexibility. The DOM allows a program to access all aspects of an XML structure, while the programming language has the full power required for manipulating this structure as needed. Scripts, and most notably Perl scripts, may also be used. But this is really programming in the broad sense. In that regard, XSLT may be seen as easier to use, although not always as powerful. In particular, its declarative approach has proven more

convenient in many cases. But all those languages make use of a textual syntax, both for representing structures and for specifying transformations.

A graphical representation of structures is often easier to grasp. Some tools have shown the benefit users take from a visual perception of structures. In the document transformation area, tools such as XML Spy [4], IBM's Visual XML Tools [20] and eXcelon Stylus [1] are good examples. They provide a graphical environment, building on top of languages such as XSLT, along with useful features such as syntax highlighting, a graphical view of source documents (e.g. using a Java JTree) and some debugging facilities. But the transformation itself is still specified textually, often in XSLT, thus requiring the user to learn the language. These tools are therefore analogous to integrated development environments such as Visual C++, which in fact rely on a textual programming language that the user has to learn. Therefore, even with some graphical help, this approach is not really satisfactory, and other paths to document transformations are worth being explored, in particular visual programming.

1.3 Visual programming

The main difference between visual programming languages and textual programming languages is multidimensionality, i.e. the use of more than one dimension to convey semantics [7]. Textual languages often use a two dimensional syntax, but only the first dimension conveys semantics, the second one being limited to documentation and increasing the legibility of programs through indentation and new lines. Examples of multidimensionality in visual programming languages are the use of interconnected graphical objects (diagrams) or spatial relationships to convey semantics.

A lot of attempts to create general-purpose visual programming languages have failed (except for some of them such as Prograph [13]), resulting in "toy" languages demonstrating interesting features but not able to handle real-world programs. This is partly due to the fact that the concepts found in general-purpose programming languages are rather abstract and not easily mapped to graphical representations. Domain-specific visual programming languages address this issue by limiting the kind of problems that can be modelled in a given language to a well identified domain, like Lab-View. This approach does not solve all problems linked to visual programming languages, but it presents the advantage of significantly improving the *closeness of mapping* [17] between entities of the problem world and entities of the program world, resulting in programming languages which use a domain-specific vocabulary. Such languages are easier and more intuitive to use and are in some cases accessible to users without any programming skills but who are familiar with the problem domain.

2. OVERVIEW

VXT [28] is a visual programming language combined with an interactive environment, written in Java, and specifically designed for programming XML document transformations. It aims at providing high quality visual support for data structure representation and manipulation, so that the user will be freed from maintaining complex mental models of data structures, which play a central role in the transformation specification.

Transformations are specified mainly visually, following a model close to the one found in XSLT. They are source driven, and consist of a set of rules which can be cascaded. Transformations are exported as XSLT or Circus source code. Circus is a programming language specialising in structure transformations (see next section). As the two languages do not have the same expressive power, VXT provides two specification modes, one for each lan-

guage. Transformations can also be run directly from the environment, which provides debugging facilities.

The interactive environment provides other mechanisms to assist the user, like the automatic generation of pattern-matching expressions from source document fragments, the ability to test such expressions on document instances at any time directly from the environment (*progressive evaluation*, see section 6), and *constrained user interaction*, a feature preventing the user from making certain kinds of semantic errors (most syntactic errors being eliminated by the visual nature of the language). All these features rely on the underlying Circus engine.

3. CIRCUS AS A TARGET LANGUAGE

Programming languages are usually designed for programmers, but just as documents are increasingly (and partially) generated by computers, programming languages are often the generation target of other, higher level, languages. The first benefit in such an approach is the reuse of the lower level compilation process and execution libraries associated with the target language. The second benefit is that the produced code is legible by a programmer, and thus may be easily adapted and changed in order to fulfill any evolving requirement. This is unfortunately not the case in practice, as the generated code is often cumbersome, too complex and not modular enough for human reading. This observation (among others) led to various theoretical and design choices for Circus, a programming language specialising in structure transformation developed at XRCE. Most notably, Circus offers a very flexible syntax and type system together with special source code composition mechanisms designed to address the difficulty of merging machine-generated code with human-generated code.

As VXT is the first very high level (visual) language built upon Circus, it is naturally being used to evaluate and demonstrate the research work. More particularly, we expect the VXT experiments to identify cases where transformations are better described partly visually through VXT, and partly textually through source code editing. Besides these cases, positive feedback is expected from using composition operators as a way to simplify the code generated by VXT, without any extra cost from the execution point of view.

Overview. Circus features a dedicated type system that enables data modeling, precise static type checking and flexible typing discipline through subtype polymorphism. Its most innovative features are homogeneous structural filtering operations, a composition algebra, and a concurrent execution model based on Linda¹. Type control is static², but dynamic typechecking and typecast can be used in filtering operations.

Formalism. The language is built upon a formal calculus including a structured operational semantics and a formal type system. This approach allows us to describe the semantics of our composition and filtering operations in a precise and mathematical way, and also to prove some important properties of the type system, such as the "type soundness" of Circus expressions (which guarantees that well-typed expressions cannot produce execution errors, under strong execution hypotheses such as infinite memory).

Abstraction level. Circus is typically at a lower abstraction level than XSLT but higher than a scripting language (such as Perl or Python) associated with a DOM/SAX compliant library. It is comparable in some ways to XQuery [10] with its associated algebra,

¹Linda is based on a central coordination memory in which processes can either write or read/consume values through filtering operations.

²Static typechecking helps preventing programming errors and enables compilation optimizations.

but is less focused on the query part and more on control structures. Not being dedicated to XML, it can also address broader categories of transformations, like Omnimark [3] which only provides a stream based execution model and neither advanced typing mechanisms nor compositional capabilities.

Rules and filtering operations. Declarative languages often involve rules, and specific rule-application strategies. Circus rules have the general form $e_1 \# f \Rightarrow e_2$, where e_1 is any valid expression computing a value (subject), f any filter having a type compliant with e_1 , and e_2 any other expression, including rules. Filters are explicitly applied to values through the matching operator $\#$. If the operation succeeds, the right part is executed in the possibly modified context (filters can indeed extract information from the subject and store it into variables).

Execution model. Rules can be gathered inside ordered collections, combined through dedicated connectors and used inside standard imperative or functional statements. In this sense, no specific application strategy is predefined, but rule combination is made extremely flexible thanks to specialised syntactic and typing strategies. Source driven transformation models are quite natural to implement through recursive functional schemes, as in functional languages à la ML, or even as in XQuery functions, but other models can be considered as well at reasonable development costs if compared to approaches based on general-purpose languages.

XML and HTML development toolkit. Circus, as many other languages, is adapted to markup processing through a library of components, type structures, basic components and tools. Proposed tools are based on XML/HTML parsing and tree linearization. Standard general-purpose analysers are built upon components that can be reused and assembled in order to quickly customise specialised parsers. With the proposed component architecture, one can choose between various strategies in order to build the parsing structure (tree-based, event-based, etc...). Standard parsing structures are based on forests, thus allowing partial document analysis. Although basic parsing components are very fine grained, their composition doesn't raise extra computation costs (composition is a static and syntactic operation).

Document type analysis. D-TaToo (Document Type Analysis and Transformation Toolkit) processes DTDs and various XML schemas. It features analysers which produce a common internal representation, called D-T, of various document type specifications. From this central representation, several Circus transformations are provided among which (i) D-T expander (to increase the legibility of any DTD or Schema by retrieving external subsets and expanding parameter entities), (ii) XHTML document generator (to produce a browsable representation of D-Ts through a frame based, hyper-linked representation), (iii) Circus type generator (to translate the document constraints expressed in the DTD into equivalent Circus types). These types can then be reused in all Circus transformations working on documents compliant with the original D-T. Thanks to this transformation, the design of fully typechecked transformation chains is possible. Such chains guarantee that an input document valid with respect to a D-T will be transformed into a new document which is valid with respect to another D-T (it is especially important when the result of the transformation must comply with an existing schema). Note that subtype mechanisms make it possible to graduate the precision of type control at the various involved levels, so that soft typing disciplines can be adopted by programmers when better suited to their work.

Examples. XML nodes are modelled through a recursive record type in which the *label* field contains the name of the node, the *sub* field contains an ordered sequence of sibling nodes and the *attr* field contains a dictionary of attribute/value pairs. Note that leaves

of such trees are strings (PCDATA items).

```
type XTree = String |
<label: String, attr: {String: String}, sub: [XTree]>
```

A compliant tree instance is for example given by

```
Mail: XTree = <label='mail', attr={'ref'='ep47'},
sub=[<label='date', sub=['26-01-2001']>, ...]>
```

and a filter able to extract the **date** element's content:

```
<label=%'mail', sub=[<label=%'date', sub=[?s]>]++?>
```

Note that this filter is constructed similarly to values, and is made of sub-filters such as `%'mail'`, which checks if the name of the root node is of type **mail**, and `?s` which stores the content of the date into variable `s`. `[...]++?` states that **date** must be the first element in the sequence of children and can be followed by others. Similarly, `?+ [...]` would express that **date** must be the last element in the sequence with zero or more other elements before it, and `?+ [...]++?` that **date** can be anywhere in the sequence.

Status. Circus is not yet publicly available, nor published because patents are currently being processed. A first version running upon a Python Virtual Machine (or a JVM after a separate additional compilation stage) is fully operational and used at XRCE.

Future work. A more advanced version including type extensions in order to support a better DTD and XML schema mapping, as well as additional filter constructors, is under development. D-TaToo is today focused on DTD, but is currently being extended to support XML Schema and possibly other schema languages (Trex, Schematron,...). Based on our central format, other converters could be considered in the future, among which a D-T instance generator (to propose a (possibly parameterized) set of document instances compliant with a D-T), a D-T learner (to induce a D-T among a set of instances), and D-T to D-T transducer generator (this difficult problem could be partially addressed by proposing a set of components as a "user-refinable" solution).

4. REDUCING THE USER'S COGNITIVE LOAD: SOURCE DOCUMENT VISUALISATION

The main purpose of source structure visualisation is to reduce the user's cognitive load by providing him with a representation of source structures so that he does not have to maintain a mental model of them. Both XML instances and DTDs can be displayed using a single visual representation system focused on tree structures, and whose goal is to simplify the complexity brought by the three abstractions manipulated in the application (XML documents, DTDs, and transformations rules) by underlining their basic unity. In VXT, visual representations of source documents and DTDs act as "background images" which can easily be panned and zoomed. Although they can help build and quickly test pattern-matching expressions, transformations are completely independent from them.

Some IDEs for XML provide a graphical representation of source structures, ranging from Java JTrees to more elaborate representations using node-link diagrams such as the one found in Near & Far Designer [2]. Although useful, these representations have several drawbacks. They do not scale very well to big documents (poor or lack of zooming and navigation capabilities, representation formalism not always well adapted). Moreover, source documents and transformations are completely decoupled (represented textually in separate frames), requiring the user to make a more important effort to mentally link the source structure to transformation rules.

```

<mail ref="ep47">
  <date>26-01-2001</date>
  <recipient>vion-dury@xrce.xerox.com</recipient>
  <sender>pietriga@xrce.xerox.com</sender>
  <subject>Jazz</subject>
  <textbody>
    <p>Hello Jean-Yves,</p>
    <p>
      Have you listened to
      <cite>My Favorite Things</cite>
      by John Coltrane?
    </p>
    <p>Emmanuel</p>
  </textbody>
</mail>

```

Figure 1: A mail document in XML

4.1 Textual and node-link representations

Even if line breaks and indentation bring some clarity to textual representations (Figure 1), which are one dimensional, graphical representations of tree structures seem to be more easily processed by users. Aside from being multidimensional, they make more information explicit.

In Figure 2, a standard node-link representation of the above document, the tree structure should be more easily understood for the following reasons: (i) relations between parent and child nodes are made explicit by the lines linking them, (ii) the shape of nodes reflects their type (element, attribute or text), (iii) the role played by syntactic constructs such as tag delimiters in text representations is taken on by other dimensions such as layout and borders, (iv) because of the multidimensionality of this representation, elements do not require both opening and closing tags, but only one identifier. All these factors contribute to speeding up the user's information processing.

4.2 Hierarchies represented as nested elements

Aside from standard node-link representations, hyperbolic and cone trees [30] (which are well-suited for data visualisation but not for its manipulation), another graphical representation of tree structures, sometimes referred to as *treemap*, consists of representing nodes as rectangles and nesting child nodes in their parent. Schneiderman [22] used treemaps to visualise large hierarchies, such as file systems. In tree representations using node-link diagrams, more than 50% of the pixels are part of the background. Treemaps adopt a space-filling approach, thus making a better use of the available screen real estate, but require zooming capabilities, since deep nodes can be very small.

Principles. VXT's visual interface relies on the Visual Transformation Machine (VTM), a Java zoomable user interface toolkit. This toolkit allows smooth zooming/navigation in infinite universes (virtual spaces) that are observed through cameras, thus making it possible to use treemaps in our application. The chosen representation is a variation of treemaps, illustrated in Figure 3. Children of a node are nested on one line, following a horizontal flow. Attributes (represented as triangles) are not considered as true children of elements and are therefore laid out separately above their bottom edge.

Representation strategies. The width of elements depends on the number and width of their children, but all elements at the same absolute depth in the tree have the same height. We think this property improves the user's perception of the structure. Indeed, in typical node-link representations of trees, the absolute depth of a node is easily perceived since all nodes at the same depth have the same

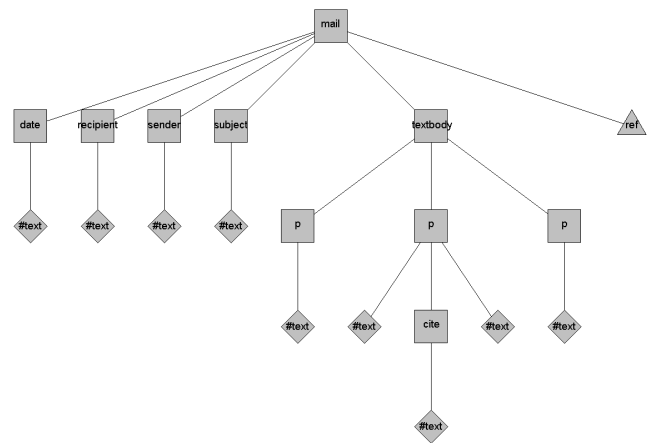


Figure 2: Node-link representation

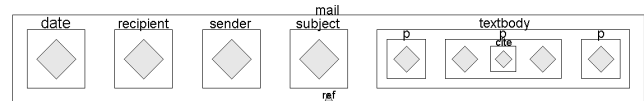


Figure 3: Treemap representation

vertical position. In treemap-like representations, this information is not so obvious, since it is only conveyed by the level of nesting of the shape. By making all nodes at the same depth be of the same height, we represent the depth information in a more intuitive way, thus improving the overall perception of the structure. We first tried another variation in which both the width and height of elements at the same depth were equal to the maximum width and height required at this depth (i.e. the largest ones). This method wasted a lot of screen real estate when displaying tree configurations where the number of children varied too much among nodes at the same absolute depth, making such trees illegible (nodes could have very small children with respect to their own size). The chosen method still has a tendency to expand horizontally, but this drawback is minimised by the continuous zooming, navigation and search capabilities provided by the environment. To address this problem, we also experimented with another way of laying out nodes inside their parent, by positioning them in a square-matrix-like way on several rows. Although more efficient in terms of pure screen real estate use, this method had the major drawback of introducing discontinuities in the flow of sibling nodes: nodes at the beginning (resp. end) of a row were far from their left (resp. right) siblings. Also, because nodes were vertically surrounded by others which were not their direct left or right sibling, the ordering of elements was not properly conveyed.

XML documents and DTDs. Although XML documents and DTDs are not at the same level of abstraction, we have tried to make their respective visual representation formalisms as close as possible, to make it easier for the user to switch between them. They both use the same representation for basic nodes, defined by the following mapping of perceptual dimensions [5]: the node's type (qualitative information) is represented by the graphical object's shape and hue (HSV colour space), and its depth (quantitative information) by its height. DTDs require additional constructs to represent the notion of sequence, choice and cardinality of elements. Figure 4 shows a DTD for the e-mail in Figure 1 and the

```

<!ELEMENT mail (date,recipient,sender,
subject,textbody)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT recipient (#PCDATA)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT textbody (p)+>
<!ELEMENT p (#PCDATA | cite)*>
<!ELEMENT cite (#PCDATA)>
<!ATTLIST mail ref CDATA #REQUIRED>

```

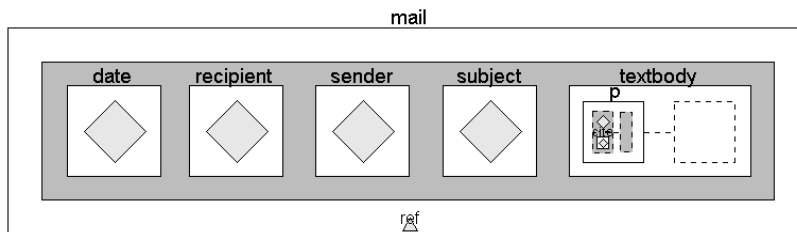


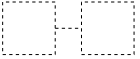
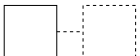




Figure 4: DTD for mail documents and its zoomable visual representation

corresponding visual representation. Sequences of nodes are visually represented by nesting the nodes in a blue rectangle (e.g. around date ... textbody), while alternatives of a DTD choice are represented by green rectangles stacked on top of one another, as illustrated in p elements. The second rectangle on the right of this choice denotes cardinality: the three occurrence indicators governing whether an element, sequence or choice should occur *one or more* (+), *zero or more* (*), or *zero or one* (?) time are visually represented as follows:

- (?) the item's outline is dashed: 
- (*) the item's outline is dashed and  is added on its right side: 
- (+) the item's outline is solid and the same symbol is added on its right side: 

The  symbol represents the possibility to have zero or more occurrences in addition to the first one. We first tried another representation using three similar and overlapping icons (). Such representations are commonly found in graphical file managers, and also in Blackwell's visual representation of regular expressions [25]. This representation consumes less space, but gives the impression that additional occurrences propagate along the z-axis (at right angles to the screen), whereas they do so along the x-axis from left to right. Moreover, overlapping icons are less distinguishable than the chosen representation in a zoomable graphical environment. For more details on the visual representation of DTDs, refer to [27], a full paper dedicated to its formal study.

The perception of the structure is significantly modified with respect to node-link representations. The nesting of nodes and the continuous zooming capabilities offer the user an intuitive way of specifying the level of details he wants when exploring the structure. Indeed, depending on the camera's altitude, the user will see nodes down to a given depth³, the nodes below this depth being too small to be displayed. Additional details can be obtained simply by zooming in, while zooming out gives more context. Moreover, treemaps combined with good navigation, zoom and search capabilities should make the representation more scalable, i.e. able to handle large documents.

Implementation. The visual representation of XML instances is automatically built from their DOM representation, obtained through a standard XML parser, while the representation of DTDs

³All nodes at the same absolute depth have the same height and the width of a node is equal or superior to its height.

relies on the D-T generated by the D-TaToo Circus module mentioned in section 3.

5. TRANSFORMATION MODEL

We chose a transformation model close to the one found in XSLT mainly because of its level of abstraction. It is easy to use while powerful enough to specify a wide range of transformations, and is well-suited to visual representation (thus increasing the closeness of mapping between the problem world and the program world). Moreover, Circus being at a lower level of abstraction than XSLT, the transformation model can easily be mapped to equivalent control structures (except for some XPath [12] axes), providing a good intersection between the two languages and thus reducing the gap between the two specification modes.

A transformation program is a set of transformation rules. Each rule associates a pattern (left-hand side of the rule) and a template (right-hand side). Rules are evaluated against the source tree nodes⁴. When the pattern is matched, the template is generated in the result document. Transformation rules can be cascaded (this corresponds to `xsl:apply-templates` in XSLT), and conflicts between rules can be solved by assigning them priorities. This feature is especially relevant in Circus mode since the notion of "longest match" (computing rule priority from the match XPath expression) found in XSLT does not exist.

6. REPRESENTING PATTERN-MATCHING EXPRESSIONS AS VISUAL FILTERS

The mental model associated with transformation rules consists of filters that match structural patterns and extract data that can be reorganised in the right-hand side of the rules. We've tried in VXT to find a representation that reflects this mental model, in which both selection of nodes and extraction of data are unified in a single operation.

In XSLT, the left-hand side of template rules (XPath expressions) specifies nodes to select in the source document. Selection of the information to extract when a rule is fired is expressed by XPath expressions in the right-hand side (`select` attribute of `xsl:apply-templates,xsl:copy-of,... nodes`), along with the kind of operation to apply on this data. The selection and extraction operations are therefore separate. The approach chosen in VXT is slightly different. The left-hand side of transformation rules, called Visual Pattern-Matching Expression (VPME), is considered as a *filter* or *mask* applied on a source structure and through which some information is extracted. The selection of nodes and the extraction of data with respect to these nodes are therefore merged in a single visual expression thanks to the multidimensionality of graphical

⁴Trees are walked in a depth-first, left to right way with some exceptions due to explicit selection of nodes when cascading rules.









node to be extracted	→	filled with semi-transparent blue colour	
node not to be extracted	→	wired (transparent interior)	
direct child/parent	→	thin border	
descendant/ancestor	→	thick border	
node actually selected (contextual node)	→	green solid border	
selection constraint, node must exist	→	blue solid border	
selection constraint, node must not exist	→	red solid border	
node existence not required (only for nodes to be extracted)	→	blue dashed border	

Figure 5: Node properties

representations. The production associated to each rule is however not specified at this stage, which is only dedicated to selecting nodes and data that will be extracted, meaning that nothing is specified about how the extracted information will be transformed.

To further convey this idea of a visual filter, all *VPMEs* are expressed in one window, consisting of two transparent layers which can be panned and zoomed independently. The background layer is used to visualise source structures (XML document instances or DTDs) while *VPMEs* are expressed on the foreground layer. The interior of *VPME* nodes can be either transparent (nodes expressing selection constraints) or semi transparent (nodes to be extracted). *VPMEs* are also structured as trees, and are represented using a formalism close to the ones used to represent XML documents and DTDs. Therefore, a *VPME* resembles the source structure it is supposed to select and extract information from. This conveys the idea of a visual filter since a *VPME*, when superimposed on source nodes that would be selected by it give the impression to match “visually”.

To solve the legibility problem caused by the two superimposed layers, users have two options when creating *VPMEs*: they can either show/hide source documents in the background layer at will with a keyboard shortcut, or VXT can render them so that they do not visually interfere with *VPMEs*. This is achieved by rendering source documents using shades of grey with minimum contrast while *VPMEs* are rendered in the foreground with highly-contrasted vivid colours.

6.1 Construction

VPMEs are built mainly visually. The user first selects the type of node (identified by its shape) he wants to create in a palette of icons, and then clicks in an existing node to add the new one as its child (clicking in an empty region creates a new *VPME*). New *VPME* nodes, and even entire *VPMEs* can also be created by extracting and converting subtrees from background XML instances (the user can make a shallow or deep copy of a source node, which is converted in the *VPME* that would select it). One node, called the contextual node, represents the node that is actually selected by the *VPME*. Other nodes, which can be ancestors or descendants of the contextual node, express selection constraints, data to be extracted,

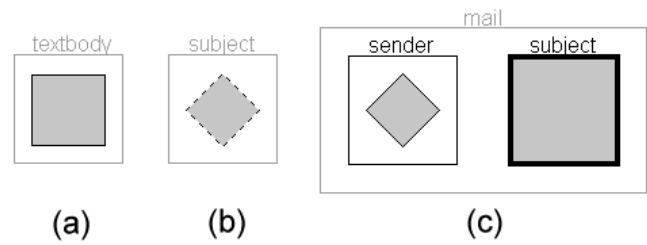


Figure 6: Examples of *VPMEs*

or both.

Selection constraints can be the existence of a child, descendant or ancestor node, or can be based on textual content. Constraints can also be negative: for instance it is possible to select a source element based on the absence of a specific node in its list of children. More complex constraints, entered textually, can select source nodes based on properties such as the number of children of a given kind.

VPME node properties are expressed using the orthogonal visual dimensions (which apply to all types of nodes, i.e. all shapes) summarized in Figure 5. Properties can be edited using a contextual popup menu or a panel. In order to reduce *viscosity* [17] (how much effort is required to edit a program), consistency checks are made whenever a property is changed, and other properties may consequently change. For instance, a *VPME* node cannot express both a negative constraint (absence of the node) and be extracted at the same time. So, if its “extract” property is set to true when making it a negative constraint, the “extract” property is automatically set to false.

This set of properties seems to provide an acceptable balance between expressiveness and complexity of the visual representation. For instance, XPath features special axes that can be used to point to nodes which are not directly accessible in the tree with respect to the contextual node, like following-sibling. Most axes are invisible in *VPMEs* since they are implicitly specified by the position and nature of nodes with respect to the contextual node (e.g. parent, descendant). These special axes, which are not essential but can simplify the user’s task, are not yet supported in VXT because of their higher abstractness, which makes them more difficult to represent.

Figure 6(a) shows a *VPME* that selects *tbody* elements (its true colour is green, as for *subject* and *mail*) on one condition: they must have at least one element child (with no constraint on its name). The grey fill colour is actually a semi transparent blue colour indicating that this information is extracted when the *VPME* matches. Figure 6(b) shows a *VPME* that matches *subject* elements and extracts their textual content. As expressed by the dashed outline, the text node(s)’s presence is not required for the *VPME* to match. Finally, Figure 6(c) illustrates a more complex *VPME* that selects *mail* elements on the following conditions: they must have a *sender* element as a direct child, and a *subject* element as a descendant. Furthermore, the *sender* element must contain some textual data, which is extracted, as is the *subject* element. It is of course possible to express much more complex *VPMEs* including constraints on the position of nodes (expressed textually) and also negative constraints.

6.2 Evaluation

It is not always easy to predict what a *VPME* will select nor to express exactly the constraints to select specific nodes. It is therefore essential for users to be able to test them against source instances

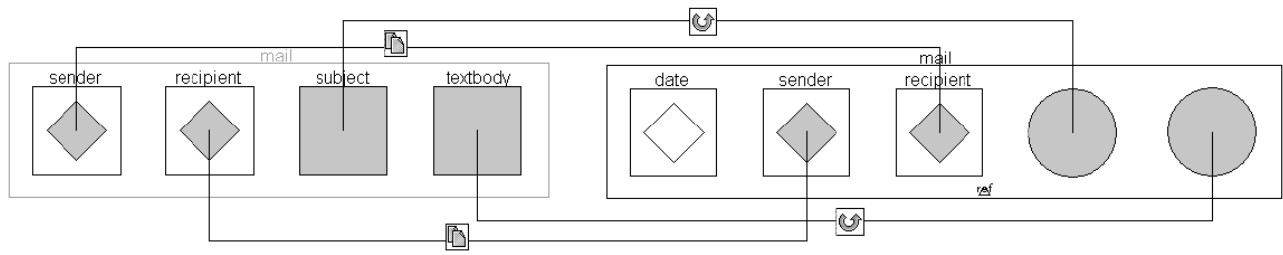


Figure 7: Reply: create a mail template

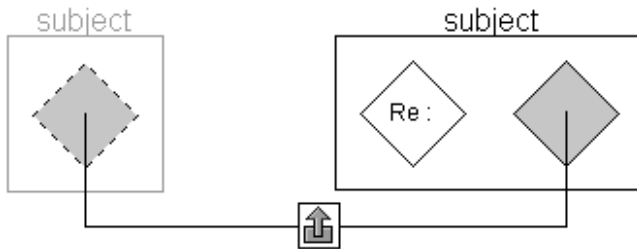


Figure 8: Reply: insert 'Re:' in Subject

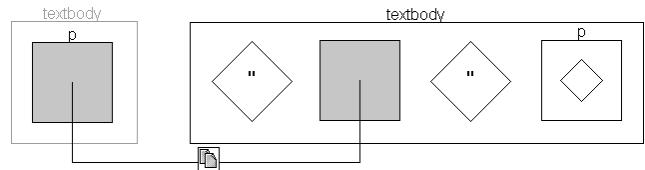


Figure 9: Reply: quote the original text

at any time and modify them accordingly without having to run the entire program ([17]-*progressive evaluation*). The superimposed layers make it possible to drag *VPMEs* on top of source document nodes in order to test whether they match the selection constraints or not, highlighting nodes which match the dragged *VPME*.

7. BUILDING THE RESULT TREE

Transformation rules being relatively independent of one another (dependencies are limited to rules calling other rules in their right-hand side), result fragments produced by transformation rules are specified in separate windows, called template windows (one for each *VPME*). This addresses part of the scalability problem [8] (the ability to handle realistically-sized problems) by creating procedural abstractions. Having all rule productions defined in the workspace used for *VPMEs* would quickly make it over-cluttered and difficult to manage. Instead, each rule is represented in its own template window, which can be opened, iconified and closed at will by the user.

A template window represents the entire rule: it contains on the left-hand side a synchronised copy of the *VPME*, and on the right-hand side the output result fragments produced when the rule is applied.

7.1 Basic instructions

As for *VPMEs*, nodes of the result-tree fragment can be created from scratch by selecting a constructor in a palette, or can be the output of a transformation operation applied to nodes extracted from the source (identified by their semi transparent blue colour in the *VPME*). These resultant nodes are linked to their source in the *VPME* by a broken line, to which an icon representing the type of operation applied is attached.

Depending on the operation and on the extracted node type, *VXT* is sometimes able to infer the output node type and gives this information to the user by assigning the appropriate shape (rectangle, diamond, . . .) to the node in the result-tree (which is otherwise rendered as a circle, meaning that the type is unknown). Available operations (and their equivalent in XSLT) are summarised below,

with acceptable input and corresponding output types.

Operation	Icon	Input	Output
deep copy of node		element attribute	element attribute
<i>xsl:copy</i> <i>xsl:copy-of</i>		text unknown	text unknown
extract text from node		element attribute	text text text text
<i>xsl:value-of</i>		text unknown	text text
apply rules to node		element attribute text	unknown unknown unknown unknown
<i>xsl:apply-templates</i>		unknown	unknown

Figures 7, 8 and 9 illustrate a transformation that prepares a reply to a mail document conforming to the DTD in Figure 4. Transformation rules are shown as they appear in each window, with the synchronised copy of the *VPME* on the left and the associated result-tree fragment on the right. The rule in Figure 7 selects mail elements under the following conditions: they must have non-empty sender and recipient elements, and there must be a subject and a textbody element (no constraint is expressed on their respective content). When this rule is fired, a new mail element is created, with a date child and a ref attribute. sender and recipient are copied from the source and inverted, while transformation rules are applied again on subject and textbody. The rule illustrated in Figure 8 selects subject elements, which can be empty as expressed by the dashed outline of the textual node. This rule produces a new subject element, with a textual node "Re:" to which it appends the value of the original text if it exists. The equivalent rules are given in XSLT:

```
<xsl:template match="subject">
  <subject>
    Re: <xsl:value-of select="text()"/>
  </subject>
</xsl:template>
```

and Circus:

```
tree # <label=%'subject', sub=[?s]> ->
y := <label='subject', sub=['Re:',s]>
```

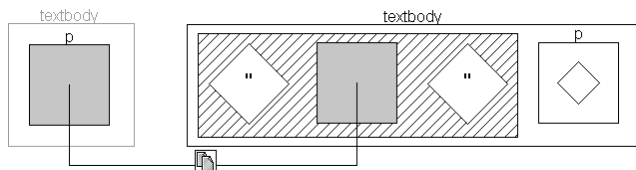


Figure 10: Reply: quote every paragraph of the original text

```
<textbody>
  <p>Hello Jean-Yves,</p>
  <p>
    Have you listened to
    <cite>My Favorite Things</cite>
    by John Coltrane?
  </p>
  <p>Emmanuel</p>
</textbody>
```

Figure 11: Reply: quote every paragraph

Finally, the last rule (Figure 9) copies the content of the `textbody` element, which is made of `p` elements according to the DTD, surrounds the set of `p` with double quotes, and creates a new `p` element in which the user will put his answer. XSLT and Circus rules equivalent to this one are:

```
<xsl:template match="textbody[p]">
  <textbody>
    " <xsl:copy-of select="text()"/> " <p></p>
  </textbody>
</xsl:template>
```

```
tree # <label=%*textbody*,
  sub=(?++[<label=%*p*>]++? and ?sq) ->
y := <label=%*textbody*,sub=["'",cat(sq[<label=%*p*>]),"']>
```

7.2 Advanced control structures

Control structures such as while- and for-loops are not easily represented in visual programming languages because of their complexity and abstraction. Some languages such as LabVIEW represent them explicitly. A more interesting approach in domain-specific visual languages consists of trying to hide loops as much as possible by integrating them with other entities in a more “natural” way. This is the approach taken in VXT for loop constructs.

Extraction instructions in a *VPME* can potentially return several nodes. They are therefore implicitly considered as potential loops. If each iteration only produces the output of the transformation operation, the loop does not need to be identified visually: as in XSLT, the `select` attribute can potentially return a forest, but an `xsl:for-each` instruction is only required when other nodes than the ones coming from the forest are associated with the production of each iteration. Similarly, explicit for-loops are simply delimited by a special-colour rectangular region that surrounds the set of nodes produced by each iteration. As an example, the *VPME* in Figure 9 extracts `p` elements from the `textbody`, and can potentially return a forest. These elements are copied in the result fragment and it is the entire set which is surrounded by double quotes, not each `p` element. If, on the contrary, the user wants to surround each individual `p` element in the result with quotes, then he just has to associate a *for-each* region (filled with an oblique line pattern in Figure 10) to the `p` element in the result containing two text nodes with quotes in addition to the `p` element itself. The result of this rule applied on the `textbody` element in Figure 1 is illustrated in Figure 11 (although this is well-formed XML, this example is unrealistic and should only be taken for what it is: a way to quickly illustrate for-loops).

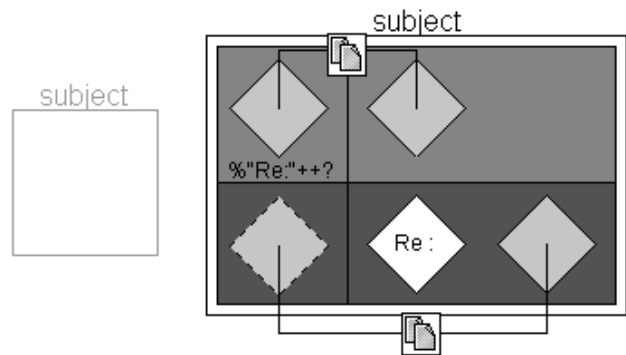


Figure 12: Reply: more complex rule for subject generation

VXT also provides conditional constructs similar to `xsl:if` and `xsl:choose`. An *if* construct consists of a condition/production pair, expressed in two adjacent green rectangles in the result-tree fragment. Conditions (in the left rectangle) are expressed as *VPMEs* evaluated with respect to the contextual node and which can contain extraction nodes, as standard *VPMEs*. The right-hand side rectangle contains a tree-fragment that is included in the result-tree only if the condition is true. As far as the *choose* construct is concerned, condition/production pairs are stacked vertically, thus expressing the idea of one choice among a set of alternatives as in the visual formalism used to represent DTDs. A darker background color is assigned to the optional default alternative corresponding in XSLT to `xsl:otherwise`. Figure 12 illustrates a rule that could replace the one in Figure 8. This rule also matches `subject` elements, but checks whether the string “Re:” is already present in the original subject. If that is the case, it just copies the original subject in the result, without inserting a second “Re:” in front of it. Expression `%"Re:"++?` is a Circus filter that checks whether a string begins with “Re:” or not. The second alternative (darker background) corresponds to the default choice. Note that the *VPME* in the left-hand side rectangle does not express a condition since the node’s outline is dashed: it is used to extract the textual content of the `subject` element (this *VPME* is evaluated with respect to the contextual node, i.e. the `subject` element).

Finally, as in XSLT, template rules can have a name which is used to manually call them from a production without having to match any node in the source document. These rules, which may not have an associated *VPME*, are accessible from a list containing all named templates, and are identified in *apply-templates* operations by their name.

8. CONSTRAINING USER INTERACTIONS

The syntax-directed environment combined with constraints on user interaction make it possible to prevent the user from making certain kinds of syntactic and semantic errors that would otherwise have been detected only at execution time. Only meaningful actions are authorized, thus guaranteeing the correctness of the specified transformations.

User interaction is constrained for both *VPME* and result-fragment construction. For instance, it is not possible to put an element node resulting from a deep copy inside an attribute or a text node. Also, depending on the specification mode (Circus or XSLT), some *VPME* constructors and transformation features are disabled to make sure that the specified transformation is compatible with the chosen language. The user is however allowed to switch between the two specification modes, the environment being in charge

of reporting incompatibilities.

9. RELATED WORK

Integrated development environments for XML such as XML Spy [4] or eXcelon Stylus [1], although often termed as visual, are not true visual programming languages, but visual programming environments in which transformations are specified textually, and are therefore very different from the solution we present. Another kind of language related to visual programming languages, are programming-by-demonstration systems, such as XSLbyDemo [23]. They are very useful for end-users who do not want to learn any programming language since they allow them to concentrate on the result (in this case presentation) without paying any attention to the programming task. The system infers general transformation rules from the user's history of actions, sometimes making inappropriate generalisations, and significantly limiting the complexity of transformations that can be expressed. XSLWiz [21], a slightly different system, allows the specification of transformations by graphically mapping fields of source and target schemas, therefore also limiting expressiveness.

On the side of true visual programming languages, XML-GL [9] and Xing [14] are languages for querying and restructuring XML data. They both define themselves as visual XML *query* languages, which makes them close to textual languages such as XQuery, whereas VXT is a visual XML *transformation* language, thus closer to XSLT. The restructuring capabilities of query languages makes the frontier between them and transformation languages blurred [24], the main difference being that query languages can express more complex queries while the transformation/restructuring part of transformation languages is more powerful (query language restructuring capabilities are often limited to grouping and sorting and do not allow cascading of rules). The same is true for XML-GL when compared to VXT, the other main difference coming from the chosen representation system: both handle XML instances and DTDs, but XML-GL uses standard node-link graphs, which are, in our opinion, less scalable.

Xing is designed for a broad audience including end-users who wish to create queries, also expressed as rules, containing information on the structure to be queried. It is therefore significantly less expressive than XML-GL and VXT. It relies on a form-based interface which does not support DTD representation and which does not seem able to handle large documents, thus limiting the tool to small transformations for end-users.

Finally, neither Xing nor XML-GL seem to provide any interactive feature to help the user in his task, like the possibility of testing queries on XML instances without running the entire program (progressive evaluation), or the display and use of DTD/source documents to help build queries (mentioned in future work, but not clearly defined). Both solutions appear to take advantage only of static features of VPLs but do not exploit any dynamic capability.

10. FUTURE WORK

A more advanced version of Circus including type extensions in order to support a better DTD and XML Schema mapping is under development (and could be extended to support additional schema languages such as TREX [11]). It will allow VXT to display all schema languages for which a Circus mapping is provided. The mapping of schemas on Circus types could also be used to explore how user interactions can be further constrained in order to produce documents that are valid with respect to a given schema, or at least valid result fragments for each transformation rule.

An interesting feature of visual environments is the sketching of

some program entities, such as the user would do on a piece of paper. This technique is not always appropriate, but some languages such as Forms/3 [18] and DocSketch [29] use it at least for some specification tasks (involving easy-to-recognise shapes). It would be interesting to allow the user to sketch *VPMEs*, primarily for specification but also when searching for an existing one (search capabilities for program entities in visual languages is an important issue). Finally, a usability analysis is planned to evaluate the usefulness of documents represented as 'background images', and the expressive power of the language (which will also be evaluated by trying to create complex transformations such as MathMLc2p [26], an XSLT stylesheet which converts MathML Content data in MathML Presentation documents that can then be rendered).

11. CONCLUSION

We have presented a visual language for the specification of XML document transformations which can be exported to two target languages: Circus and XSLT. This domain-specific programming language tries to simplify the complexity brought about by the different levels of abstraction (transformation rules, document instances, schemas), by unifying them in a single visual representation system focused on tree structures, and by providing visual metaphors close to the mental model associated to the considered transformations.

VXT is also an interactive environment in which transformation programs can be run and debugged, and which takes advantage of dynamic capabilities associated with graphical environments, such as the progressive evaluation of transformation rules, and the prevention of some kinds of errors by constraining user interaction. Finally, the chosen representations, combined with the navigation model and its continuous zooming capabilities, should make VXT suitable for specifying realistically-sized transformations.

12. ACKNOWLEDGMENTS

We would like to thank Lionel Balme, Veronika Lux and Sylvie Perriere for their work on the D-TaToo Circus library.

13. REFERENCES

- [1] Excelon Stylus, 2001. <http://www.exceloncorp.com>
- [2] Near & Far Designer, 2001. http://www.opentext.com/near_and_far/
- [3] Omnimark technologies, June 2001. <http://www.omnimark.com>
- [4] Xml Spy, 2001. <http://www.xmlspy.com>
- [5] J. Bertin. *Semiology of Graphics*. The University of Wisconsin Press, Madison, 1983.
- [6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. <http://www.w3.org/TR/2000/REC-xml-20001006>
- [7] M. M. Burnett. *Encyclopedia of Electrical and Electronics Engineering*, chapter What is Visual Programming. John Wiley & Sons Inc., 1999.
- [8] M. M. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *IEEE Computer Magazine*, 28(3):45–55, 1995.
- [9] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *8th WWW conference*, 1999.
- [10] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML, February 2001. <http://www.w3.org/TR/xquery>

- [11] J. Clark. TREX - tree regular expressions for xml, February 2001. <http://www.thaiopensource.com/trex/>
- [12] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999. <http://www.w3.org/TR/xpath>
- [13] P. Cox, F. Giles, and T. Pietrzykowski. *Visual Object Oriented Programming*, chapter 3 - Prograph, pages 45–66. Manning, 1995.
- [14] M. Erwig. A visual language for XML. *IEEE Symposium on Visual Languages*, pages 47–54, 2000.
- [15] S. A. et al. Extensible stylesheet language (XSL) version 1.0, November 2000. <http://www.w3.org/TR/xsl/>
- [16] J. Ferraiolo. Scalable vector graphics (SVG) 1.0 specification, November 2000. <http://www.w3.org/TR/SVG/>
- [17] T. Green and M. Petre. Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
- [18] J. Hays and M. Burnett. A guided tour of Forms/3. Technical Report TR 95-60-6, Oregon State University Dept. of Computer Science, 1995.
- [19] A. L. Hors. Document object model (DOM) level 2 specifications, November 2000. <http://www.w3.org/TR/DOM-Level-2/>
- [20] IBM. Visual XML tools, 2001. <http://www.alphaworks.ibm.com/tech/wsde>
- [21] Induslogic. XSLWiz, 2001. <http://www.induslogic.com/products/xslwiz.html>
- [22] B. Johnson and B. Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. *Proceedings of the 2nd International IEEE Visualization Conference, San Diego*, pages 284–291, 1991.
- [23] T. Koyanagi, K. Ono, and M. Hori. Demonstrational interface for XSLT stylesheet generation. *Extreme Markup Languages*, 2000.
- [24] E. Lenz. XQuery: Reinventing the wheel? 2001. <http://www.xmlportfolio.com/xquery.html>
- [25] H. Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*, chapter SWYN: a visual representation for regular expressions. M. Kaufmann, 2000.
- [26] E. Pietriga. MathMLc2p : Content to presentation transformation, December 2000. <http://www.inrialpes.fr/opera/people/Emmanuel.Pietriga/mathmlc2p.html>
- [27] E. Pietriga and J.-Y. Vion-Dury. A formal study of a visual language for the visualization of document type definition. *IEEE Symposium on Visual Languages and Formal Methods (Human Centric Computing Languages and Environments)*, September 2001.
- [28] E. Pietriga and J.-Y. Vion-Dury. VXT: Visual XML Transformer. *IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering (Human Centric Computing Languages and Environments)*, September 2001.
- [29] M. Pinto-Albuquerque, M. Fonseca, and J. Jorge. Visual languages for sketching documents. *IEEE Symposium on Visual Languages*, pages 225–232, 2000.
- [30] G. Robertson, S. K. Card, and J. D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4), 1993.