

XML Visualization Using Tree Rewriting

Josef Jelinek
Czech Technical University in Prague
jelinej1@fel.cvut.cz

P. Slavik
Czech Technical University in Prague
slavikp@cslab.felk.cvut.cz

Abstract

In many applications complex structured data have been used. Visualization of these data structures allows a user to get better insight both in the data structure and in the application itself. We present a visualization technique of structured data-oriented XML formats. Customizable visualization is done by means of tree rewriting. This allows for an efficient filtration and transformation of the document tree to fit particular user needs as well as a transformation to a structure with a predefined graphical interpretation. A graphical representation of the tree rewriting rules for an interactive manipulation is presented, so the system itself can be considered a graphical language. The graphical representation is based on tree maps.

Keywords: XML Visualization, Tree Rewriting, Visual Programming

1 Introduction

Many applications use complex structured data formats to store their data. Nowadays there are trends to use standardized data formats. The advantage is that an application can use existing tools and program libraries to handle such data and developers need not create their own tools and input/output modules.

XML (Extensible Markup Language [W3C]) is a standardized flexible markup language format for creating structured data for specific domains (a specific XML application). Data specified in XML are tree-structured using properly nested markup tag pairs.

XML documents can be divided into two basic types: *message-oriented* and *data-oriented* [Harrold and Means 2002].

The message-oriented XML documents contain mostly readable text and can be well legible even after all tags from the document are removed. Here the tags give only additional information how to interpret the particular

text, but are not usually necessary to retrieve the important message. Such documents can be read quite easily as plain text.

The data-oriented documents are aimed mainly at the domain of computer processed data and are not usually suitable for viewing by a user without specific visualization software that is able to interpret the particular document and present it to the user. In these documents, *tags* (markup elements) are used to create the structure of a document and to give particular interpretation to the data of the document. If the tags are removed, the remaining data would likely lose their meaning. In our work, we have focused on the data-oriented XML documents that are less readable in its textual representation and thus more challenging for visualization.

Retrieving information from the plain textual representation of data-oriented XML documents is often very hard. The text needs to be well indented to provide a particular context. However, well indented text requires quite large space. These two requirements are often solved by means of folding parts of a document tree. The folding is usually fully controlled by a user. However, it is often necessary to retrieve relevant data from more sub-trees at once and the folding technique often shows only one sub-tree at a time (the other sub-trees are either folded or out of a current view).

Another suitable technique that solves some of the previous problems is *document transformation*. There is a standardized XSLT format [W3C] for XML transformations. It specifies rules for transformation of an input XML document to another XML document. It is a powerful tool which can be used to specify almost any transformation and customization. As an example of a system that uses XSLT for graphically specified transformations we can give VXT [Pietriga et al. 2001].

However, there are several drawbacks. Because of its complexity, the XSLT is quite hard to learn and use. It also requires that the user already knows the structure of the input XML document that can require some time to learn. In addition, the XSLT is not very suitable for interactive usage.

We designed and implemented a system to transform and visualize the input tree-structure (particularly the XML document) that addresses the mentioned problems. Transformations are specified using a designed *graphical language* (also called a *visual language*) [Green and Petre 1996; Shu 1988].

Although we describe the processing and visualization of tree structures, it is possible to use the same techniques to handle general graphs with some hierarchy. We discuss this possibility and supporting features of the presented system in the latter sections.

The rest of this paper is organized the following way. Section 2 describes the proposed graphical representation of the XML trees. Section 3 shows designed document-tree transformations using tree rewriting and the graphical representation of rewriting rules. Section 4 introduces customizable visualization of transformed trees. Section 5 discusses XML-specific problems. The remaining sections give a conclusion, related work and future work proposal.

2 Graphical Representation

The graphical representation of a visualized tree structure is chosen to help even an inexperienced user to navigate in the tree structure without much additional knowledge about the used graphical notation. We have avoided flowing layouts and box-and-wire representation, because these visualization techniques are more suitable for more experienced users, and for larger tree structures it becomes either very dense (if all elements are displayed) or imprecise (if some parts of the structure are hidden or simplified). Such a representation is used e.g. in dataflow graphical languages Prograph and LabView [Green and Petre 1996]. In [Petre 1995] it has been discussed (based on various user-tests) that for inexperienced users the restricted and fixed graphical notation is more efficient because of novice user's lack of readership skills to interpret additional layout cues (if present). In addition, to layout less restricted graphical notations are hard problems to do optimally or at least sub-optimally.

2.1 Tree Visualization

Tree-maps [Shneiderman 1992] seem to be a good choice for specific tree visualization. Its graphical representation is constrained and uses nesting boxes to show the hierarchy. The amount of data visualized in a particular area is quite large compared to box-and-wire representation. However, to use a similar approach in this case, several modifications must take place. Classical tree-maps use the size of a 2D area to display relative sizes of nodes of an input tree. The size is not interesting for the data structures visualized in our system and should be reduced to display as much data as possible. The layout of nested trees should be more constrained to help a user read and interpret the information easily, especially when two trees are compared.

The implemented graphical representation uses nesting boxes to display hierarchy. The visualized tree is represented by a box with a title at the upper side that shows the name of a root node. All child elements are nested in the box below the title. The nested elements are displayed horizontally. A similar approach uses a query language Xing described in [Erwig 2000].

To simplify some constructs *lists* can be used. A list is collection of any number of elements. Each list can be divided into its first element and the list of its remaining elements. This property is important for the specification of transformation rules (described in the latter sections). Items in a list are displayed vertically. The *variables* are also important for the rule specification. A variable is an object that any value can be assigned to.

To clarify this let us look at an example. To represent trees in a textual notation we will use the '(' and ')' syntax to attach a list of children to a root, a list of items will be enclosed in '[' and ']' and a variable will be preceded by '?'. For example, a tree 'a' with the list '[1, 2 | ?x]' as its first child, and as a second child a tree 'b' with '3' as its only child would look like 'a ([1, 2 | ?x], b(3))'. The first two items of the list are '1' and '2' and the list of the remaining items of the list is expected to be the value of the variable '?x'. The resulting tree and the associated graphical representation are shown in Figure 1.

The figures originally use distinct colors to distinguish the types of displayed graphical elements. This allows more flexibility than shades of grey used in this article. However, the graphical representation is chosen to be clear even on graphical devices without a color support.

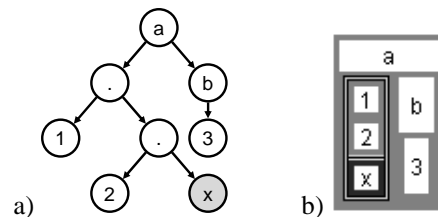


Figure 1. Representation of a tree a) classical representation b) implemented graphical representation

A variable need not be only a symbol. It is possible to use any tree or list as a variable. The concrete examples can be then shown instead of abstract symbols that need not be familiar to all users (although this approach can be more space consuming).

3 Tree Rewriting

Tree rewriting is a model of computation which is used in a variety of applications within computer science. Although it is rarely used as a base for programming languages, there are some attempts. One of the earliest attempts is an *equational logic* [O'Donnel 1985]. Another more recent attempt is the implementation of the *Aardappel* language described in [Oortmessen 2000]. In addition, already mentioned XSLT is based on tree rewriting.

Tree rewriting is a very simple and orthogonal way of transformation specification – everything is tree-structured. It is closely related to graph grammars and graph/icon rewriting. To write a program you define a set of rules, each of which say: “if anywhere you find a (sub) tree that has this shape then replace it with a tree of the

following shape". If you give the program a tree then the program will try to apply as many rules as possible until the tree is rewritten into a shape to which no rules apply (this is called a *normal form*).

3.1 Tree Rewriting Properties

Tree rewriting is general enough to express any algorithm to perform on a given tree structure. It is very similar to the lambda calculus in terms of simplicity and expressive power.

The implemented version of the tree rewriting system uses similar constructs as functional programming languages based on lambda calculus. The design decisions of the system were influenced by similar decisions in the designs of the existing functional languages, mostly those that affect efficiency of the implemented system. The main features are:

- applicative (greedy, bottom-up) order of rule applying – the input tree is processed from its leaf nodes to its root node
- built-in conditionals (*if*, *or*, *and*) to omit unnecessary computations in one of then or else-branches

However, the explicit *if*-conditions are not used so often as in functional languages, because most of conditions and branching is handled by the rule selection (using rule-head matching).

The implemented rule selection strategy tests applicability of rules in the "source-code" order. It means that the list of rules is processed from the beginning and the first rule which matches (its head matches) the given tree is selected and used to transform it to a new tree.

The result of such design decisions is an efficient computation and possibility to compile the rules to a sequence of primitive imperative language statements. As an example, the *Aardappel* language [Oortmerssen 2000] uses Java virtual machine bytecode as a target for compiled rules as well as the system presented.

3.2 Graphical Notation

The required transformations can be described by transformation rules. Graphical representation of the rules for tree rewriting utilizes human ability of pattern recognition of a fixed graphical structure.

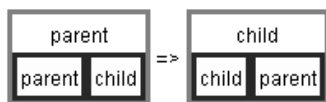


Figure 2. Rule example

Very simple example is shown in Figure 2. The meaning of the rule is that anywhere in the input tree the sub-tree *parent* with two child elements is found the occurrence is replaced by a tree *child* with swapped child elements. In the rule both child elements on both sides are variables and are used to transfer a particular value

from the original tree to the new one. There can be more rules for transformation of the same (sub) tree and the first that matches is applied.

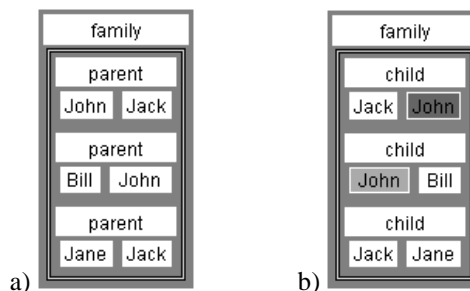


Figure 3. Transformation example a) input b) output with the same selected objects highlighted

The Figure 3 shows the transformation of a simple tree using the rule in Figure 2. One displayed feature is worth a note. To help a user recognize all dependencies between (otherwise unrelated) sub-trees, a user is allowed to select an object in the structure and all occurrences of the same object are automatically highlighted. It seems to be a good alternative for displaying such dependencies using connecting lines (which is quite common in other graphical languages) that often leads to unreadable "visual spaghetti" of many crossing lines (see [Green and Petre 1996] for examples). The reason is that as long as there are dependencies across the tree-hierarchy, the resulting structure is not a tree but a general graph. If a general graph is not planar it cannot be displayed without line crossing. Even displaying complex planar graphs is not always possible (e.g. using straight lines) or suitable (if topology or layout is important).

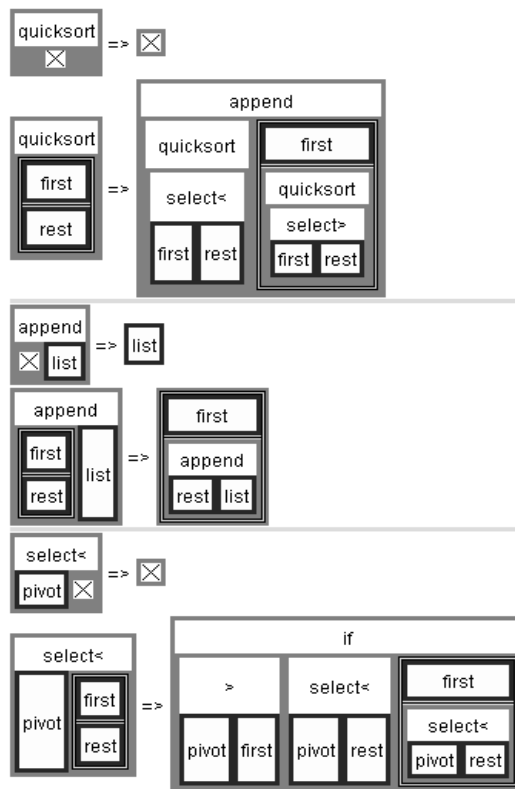


Figure 4. Graphical quicksort algorithm example

Displaying the object dependencies is also possible in rules. This feature is important especially for highlighting the corresponding variables on both sides of a rule.

To show the flexibility of tree rewriting in combination with graphical representation of the rules the example in Figure 4 shows a part of the *quicksort algorithm* to sort the input list. Rules for `select>` are left out and are analogical to those for `select<`. The “cross” symbols in the box mean an empty list. Since it is only demonstration the efficiency is not the best possible.

```

<mail ref='af62'>
  <date>2/25/2004</date>
  <recipient>joe@mail.dum</recipient>
  <sender>jelinej1@fel.cvut.cz</sender>
  <subject>Look at that</subject>
  <textbody>
    <p>Hi Joe,</p>
    <p>
      look at the
      <a href='http://page.dum/index.html'>
        http://page.dum/index.html
      </a>.
      Isn't it <i>cool</i>?
    </p>
    <p>Josef</p>
  </textbody>
</mail>

```

Figure 5. A sample XML document

Let us consider a simple example of an XML document in Figure 5 describing a mail message. It is not a pure data-oriented XML. However it has a clear structure and the main problem of this example could be nesting tags in text (the second paragraph in the `textbody` element). Such mixed content is handled as plain text and the inner structure is not considered. In the case when an element contains textual data, they are usually data to be shown to the user and the nested tags are used only to provide the text with some additional attributes that are not necessary for interpretation.

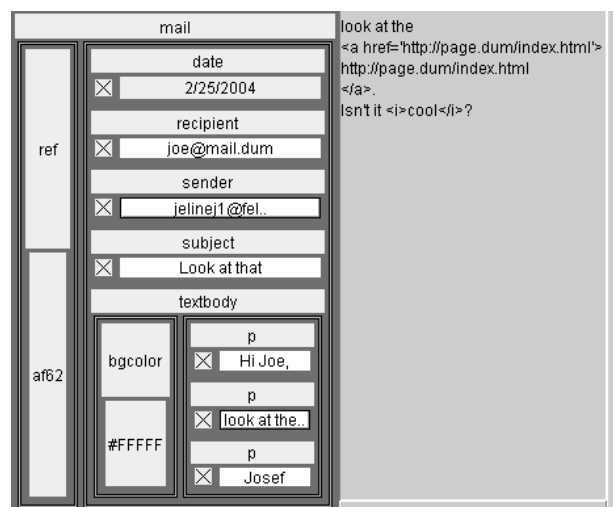


Figure 6. Graphical representation of the example XML

The textual data (eventually with nested tags) are shown either in the graphical tree (if they are reasonably small) or in an adjacent “detail” window. The graphical representation of the XML example is shown in Figure 6.

To clarify the graphical representation, several elements should be transformed and/or filtered. For example attributes of the elements are not important in this case and we can remove them all. In addition, “sender” element is not important and we can delete it. We also do not need names of some elements. Simple rules are shown in Figure 7.

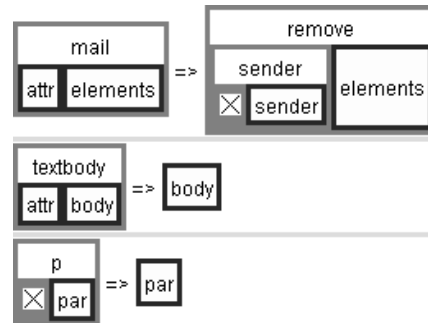


Figure 7. Rules to transform the example XML

The first rule transforms the entire mail element to the list of its child elements and removes a sender element. The second rule transforms `textbody` element to the list of its child elements and its attributes and name are removed. Similarly, the last rule transforms ‘p’ elements. The transformed graphical representation is shown in Figure 8.

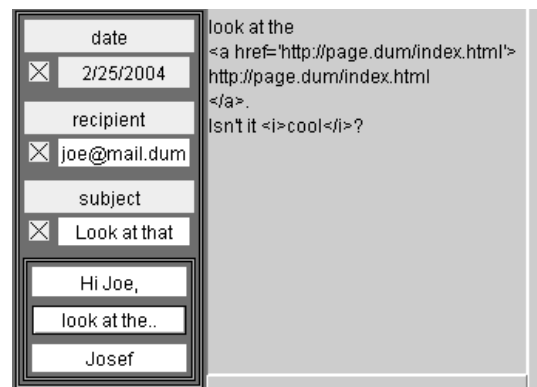


Figure 8. Transformed graphical representation

4 Customizable Visualization

The transformations and visualization described in the previous section are still too restricted. The layout and color are predefined and unchangeable and the visualization tries to be as general as possible. Since the transformation using the described rules can create new trees and elements, a solution to this problem is associating a predefined interpretation to such newly created trees. The similar idea is used in the *XSL formatting objects* (XSL-FO) standard [W3C] to provide an interpretation of elements of typesetting systems.

To transform our previous example into a more specific graphical representation, we can use built-in primitive formatting commands such as 'hbox(list)' or 'vbox(list)' to specify the horizontal or vertical layout of items in the list. Another built-in formatting command that we use here is 'box(title, color, element)'. This command is used to specify a background color and to add a title (if specified).

Let us suppose that we want to format the example as follows:

- all elements are aligned vertically except for content of header elements
- header elements have darker color than body elements
- header elements are transformed to more usual notation "name: value"

The rules for such a transformation are shown in Figure 9 and the resulting transformed and customized example is shown in Figure 10.

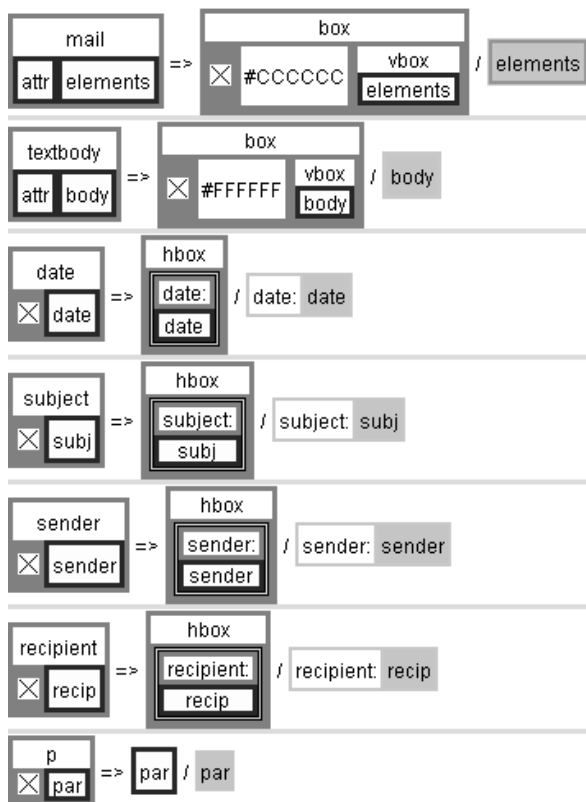


Figure 9. Rules for customized visualization

Rules using customized formatting are extended by another field on the right-hand side. This box shows graphical interpretation of the transformed tree, so user can edit (using an extended set of tools) the rules and see the intermediate transformations that are done by the specified rules. The representation need not be exactly the same, because the visualization is context sensitive and the rules do not have a particular context and can be used to transform trees in different contexts.

Other useful predefined built-in formatting commands can be used to specify binding to the associated detail window, addition of hyperlinks, different layout

managers (distribution of elements according to a visualized area), folding sub-trees, etc.

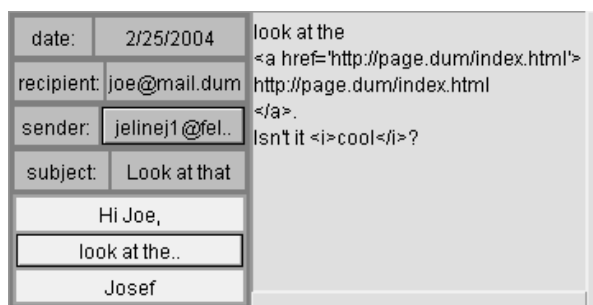


Figure 10. Transformed and customized visualization

The important advantage of the presented system is that the user can display an XML document without any knowledge about its internal organization and then incrementally add rules to transform and/or filter specific parts of the document in its graphical representation. The editing of rules is done also graphically using drag and drop and the user needs to write only the names of elements or text to appear if there is not any place where it is already used to drag and drop it to the required location.

4.1 Use case

As a more practical example, suppose that we obtained XML encoded data files describing for example benchmark results from a benchmark program. Because the format is not documented and we may not have an access to the benchmark program, we have limited possibilities to compare several results. If we try to visualize the internal structure, it is too complex to get the relevant data from more documents. An example visualization of raw data is shown in Figure 11 and takes considerably large amount of space that is roughly equal to the plain text representation (note that only about 20% of the content is shown).

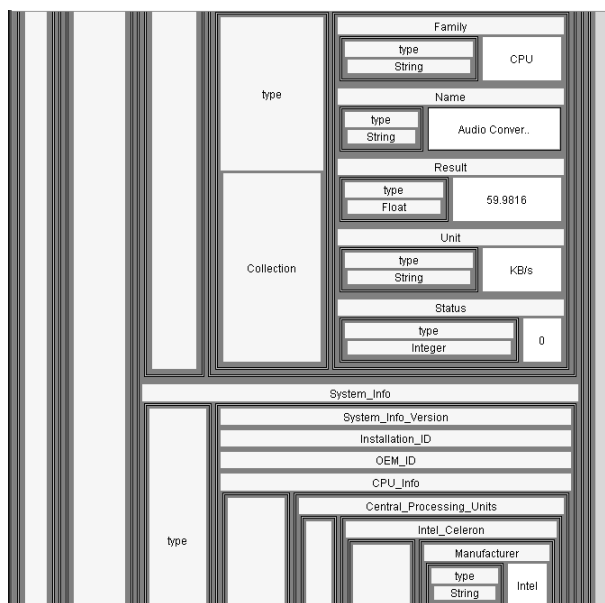


Figure 11. Visualized part of raw data file

To visualize this document in a more compact way, some simplifications should be used. Finding an important spots in the tree structure of the document can determine which sub-trees should be transformed to be visualized and which should be filtered out. For example, we can ignore information about the benchmark program and versions and we can focus on overall CPU information and speed. The element attributes specifying the type of content can be ignored for relative comparison. However, they can be useful for specific transformations (ambiguity of some elements).

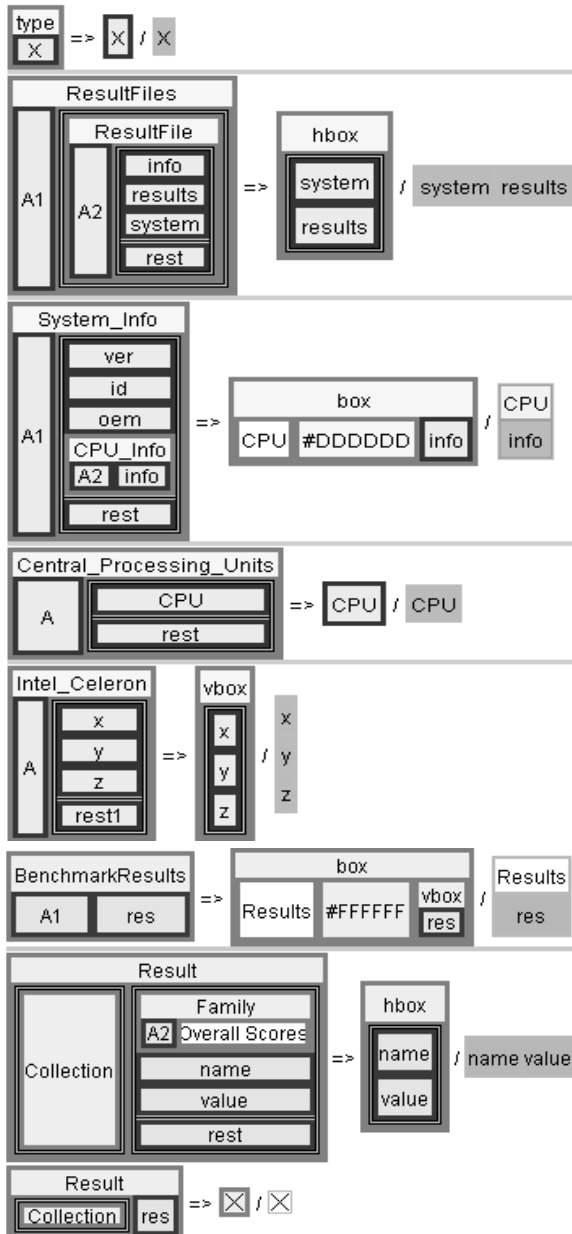


Figure 12. Rules to transform a benchmark file

The defined example rules shown in Figure 12 follow these requirements and the number of the rules is quite small (8 in this case). The resulting transformed and customized graphical representation containing mostly required data is shown in Figure 13.

Although the graphical representation of the transformed tree structure contains redundant information (types of

values, empty lists in results) it is not too disturbing for our purposes and it can be filtered out using some additional rules if necessary. The rules can be added incrementally as the user recognizes important or unimportant parts of the structure.

CPU		Results	
Manufacturer		Name	Result
String	Intel	CPU Score	Integer 3081
Family		Name	Result
String	Celeron	Memory Score	Integer 1508
Internal_Clock		Name	Result
String	1,28 GHz	HDD Score	Integer 996

Figure 13. Transformed benchmark file representation

This example shows the power of the presented system. Using only eight visual and relatively simple transformation rules the input document was transformed into a very compact digest of required data. In addition, these rules can be used to visualize other documents with the same (or similar) structure. If other information appears in these documents additional rules can be specified to handle them.

5 XML Specifics

There are several simplifications of handling XML documents in the implemented system. For example, we do not take an advantage of a DTD (*document type definition*) often associated with XML documents. One reason was that we worked with XML documents without DTDs. Supporting DTDs can bring some additional possibilities both in the visualization and in the rule editing.

Namespaces are another XML feature that is not fully supported. The namespace prefix is handled as a part of name and is separated by a colon character. Supporting namespaces is not crucial for visualization. However, as soon as the system is required to be extended to transform the XML into another document the namespace support could be necessary. In such a case using another more appropriate application (e.g. some XSLT transformer) can be used.

The presented graphical representation of the tree structure of an XML document is quite simple. It uses binary tree composed of elements that has a list of the element attributes as the first child and a list of the nested elements as a second child. The name of the element is displayed as a parent of these two children.

Other representations are also possible. There can be more general ones that use tree nodes with more children and use more general node names (e.g. `element`). There can be also used trees enclosing the children and determining their content (e.g. `'name(name)'`),

'attributes(list)', etc.). Such more general representations would have an advantage that transformation rules would be more flexible and general, but the raw graphical representation would be more complex and inefficient. In the supported representation the generality of the rules can be reached using built-in functions to get information about tree, e.g. to retrieve the node name and then process it as a string.

In the current state, better results are obtained from XML documents that contain elements with few attributes. Some of the current XML documents are defined to contain most of information in attributes, but it often leads to more complicated parsing and processing.

6 Related Work

Similar graphical representation is used by Xing [Erwig 2000]. It uses very simple tree-maps variant to visualize the XML structure. However, it is designed to be a query language for XML and it is not focused on structure visualization and/or transformation.

Another system that is aimed at transformations of XML documents using graphical rules is VXT [Pietriga et al. 2001]. However, it is designed to be mainly a graphical replacement of XSLT (or similar) language to make the general XML transformations easier to specify and it is not aimed at the interactive incremental transformation and visualization of the structure.

The most related approach of tree rewriting system is the Aardappel language [Oortmerssen 2000], which uses graphical representation of rewriting rules. However, the graphical representation is more oriented to programming and the tree structure is not so explicitly visible in its graphical representation. In addition, it is designed as an experimental general programming language.

7 Future Work

The possible extensions of the system are mostly on XML processing part. Using DTDs to assign certain properties to visualized tree (based on the element type defined in DTD) can clarify the resulting appearance. The DTD can be also used to check validity of defined rules.

Another approach to visualize a mixed content (text and tags on the same level) of an element should be considered. In the current state, the mixed content is handled as plain text (including any nested tags) and displayed in an associated detail window.

The implemented tree rewriting subsystem can be extended beyond the concrete application to visualize tree structures and it would be possible to extend it to a more general graphical programming language.

Extending the visualization to the three-dimensional space is worth considering. In this case, the visualization of the tree structure and the manipulation with it are not so close to the current graphical user interfaces and lead to a different editing style and problems.

8 Conclusion

A system that implements the presented transformation and visualization techniques was implemented as a Java application.

The presented tree visualization is based on tree-maps, so the dimensional graphical representation is compact and uses nested boxes to display the hierarchy. The dependencies across the tree hierarchy are displayed using highlighting all spots where the same object (variables, etc.) as the one selected by a user is used. The graphical representation does not use lines connecting the related spots that can lead to "visual spaghetti".

To transform and/or filter the displayed structure the tree rewriting rules are used. Rules are displayed and edited graphically using drag-and-drop. Visualization of the tree transformed by the specified rules can be done almost immediately and the rules can be defined incrementally.

The graphical representation of a tree can be customized using built-in formatting commands that modify the appearance of a certain tree structure. The transformation rules are used to transform the selected tree structure to the structure that uses these formatting commands. The commands can affect visual appearance of trees as colors, layouts, borders, etc.

The advantage of using defined rules to transform the visualized structure compared to a direct manipulation with the structure is that the defined rules can be used to transform other documents with the same (or similar) structure.

The designed system has several limitations. If the visualized data are large and homogenous (it contains many elements of few types at the same level) the raw graphical representation can degrade to a long sequence of similar boxes. However, a rule that spread the boxes more evenly can be used. Another limitation common to most visualization techniques is a limited size of screen for larger data. This can be partially solved using a sub-tree folding. The folding can be driven manually (using mouse) or by defining a rule to perform some initial folding.

References

- ERWIG, M. 2000. A Visual Language for XML. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, 47-54.
- GREEN, T. R. G., and PETRE, M. 1996. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing* 7, 2, 131-174.
- HARROLD, E. R., and MEANS, W. S. 2002. *XML in a Nutshell Second Edition*. O'Reilly&Associates, Inc.
- O'DONNELL, M. J. 1985. *Equational logic as a programming language*. MIT Press.

- OORTMERSSEN, W. v. 2000. *Concurrent tree space transformation in the Aardappel programming language*. PhD Thesis, University of Southampton.
- PETRE, M. 1995. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming, *Communications of the ACM*, 33-44.
- PIETRIGA, E., VION-DURY, J.-Y., and QUINT, V. 2001. VXT: A Visual Approach to XML Transformations. In *Proceedings of the 2001 ACM Symposium on Document Engineering*, 1-10.
- SHNEIDERMAN, B. 1992. Tree visualization with tree-maps: 2-d space-filling approach. In *ACM Transactions on Graphics (TOG)*, 92-99.
- SHU, N. C. 1988. *Visual Programming*. Van Nostrand Reinhold.
- W3C. *Extensible Markup Language (XML)*.
<http://www.w3.org/XML>
- W3C. *XSL Transformations (XSLT) Version 1.0*.
<http://www.w3.org/TR/xslt>
- W3C. *Extensible Stylesheet Language (XSL) Version 1.0*.
<http://www.w3.org/TR/xsl>