GENVIS - Model-Based Generation of Data Visualizers

Ansgar Bredenfeld
GMD - Institute AiS
Schloss Birlinghoven
53745 Sankt Augustin, Germany
bredenfeld@gmd.de

Edmund Ihler, Oliver Vogel
SYSTOR AG
Peter Merian-Strasse 84
4002 Basel, Switzerland
{edmund.ihler,oliver.vogel}@systor.com

Abstract

The tool GENVIS supports the integration of an XML data repository with different GUI frameworks. It maps a model of an XML data repository to an abstract grid-based layout model. This mapping on model level combined with a flexible template mechanism allows to generate source code for data visualizers which construct well-formatted views on XML documents. GENVIS generates data visualizers for different GUI frameworks. The tool is productively used to construct report visualizers for a credit risk report management system of a leading Swiss Bank. GENVIS was developed with the prototyping environment APICES. Since large parts of the tool were generated from meta-models of the repository and the layout, we were able to construct it at reasonable costs and effort in a very short time.

1. Introduction

System integrators face the problem of combining several sub-systems and frameworks from different manufactures into a single integrated system. In this paper, our example is a report management system which consists of authorization data sources, an off-the-shelf document management system, and web-based user interface frameworks. Since the sub-systems to be integrated are often very different in nature, language, and platform, system integrators can not solely rely on integration standards like XML or frameworks like CORBA to solve specific integration problems. They have to solve their problem individually with additional manually written glue software. Those scenarios are typical of system integrators and highly related to the problem of framework integration [1].

We identified basically two fields of problems in our integration scenario. The first one deals with the integration task itself. We have to integrate an XML report database with the HTML-based visualization framework of the document management system LiveLink [2]. The report viewer has to be written in OScript, the extension language of LiveLink. It accesses the database storing XML documents and dynamically creates HTML code. Before our project, the source code of this data visualizer has to be written manually in OScript. It had to be modified for each change in the structure of the XML repository, for each change in the visual appearance of a report and - this is very important - for each further software releases of the report management system LiveLink. All these changes were cumbersome and error-prone. Moreover, we do not want to support visualizers for LiveLink only but also for standards like XSLT [3], and in addition two SYSTOR-internal Java-based visualization frameworks. These are JWeaver [4] - a Java servlet-based framework for pure HTML visualization - and CRA (Client Reference Architecture) [5] which supports Java thin clients and allows meta-data defined user interfaces.

We tackle our framework integration problem by building the new tool GENVIS. It allows to generate report visualizers automatically from abstract specifications of the repository and the layout.

The second topic covered in this paper is the construction of such integration tools. We have to automate the tool construction as far as possible in order to make it cost-effective. This is an important issue because glue software required to solve specific integration problems is obviously not a mass product. The cost of tool construction can not be distributed to thousands of product instances, as they are typical of any general purpose software. Therefore, the second major point in this paper is the automated construction of GENVIS using the rapid prototyping environment APICES [6].

The paper is structured as follows. Section 2 gives an overview of the meta-modeling approach that we used to solve our integration problem. Section 3 describes the design of the tool GENVIS with the prototyping environment APICES. Section 4 reports on experience with GENVIS and its making. Section 5 puts our work in perspective to related work. We end with conclusions and an outlook to future work.

2. The model of GENVIS

We tackle our framework integration problem on the highest possible level of abstraction. We reformulate it as a mapping of elements in a tree-structured repository to cell contents in a grid-based layout. Therefore, we first have to abstract from the data source - a repository containing XML documents - and second from the visual target - a grid-based cell-layout displaying documents contained in the repository. Subsequently, we specify a mapping of the source model to the target model. This mapping is exploited to control the code generation for data visualizers.

2.1. Repository model

The meta-model of the repository only supports two types of elements - leaf elements and composite elements. Leaf elements are single or list data fields of basic data types, for example integer, string, date, currency, or similar data types. Composite elements are arbitrary hierarchical aggregations of leaf elements or composite elements. These meta-model notions are formally described by a document type definition. We call this description *meta-DTD*. Leaf elements have the tag name LEAFELEM, composite elements the tag name COMPELEM. The model of a concrete repository is now specified by a *meta-XML description* which of course has to conform to the meta-DTD. Each considered type of report or - speaking in terms of databases - each possible database schema of the repository can be specified by a meta-XML description.

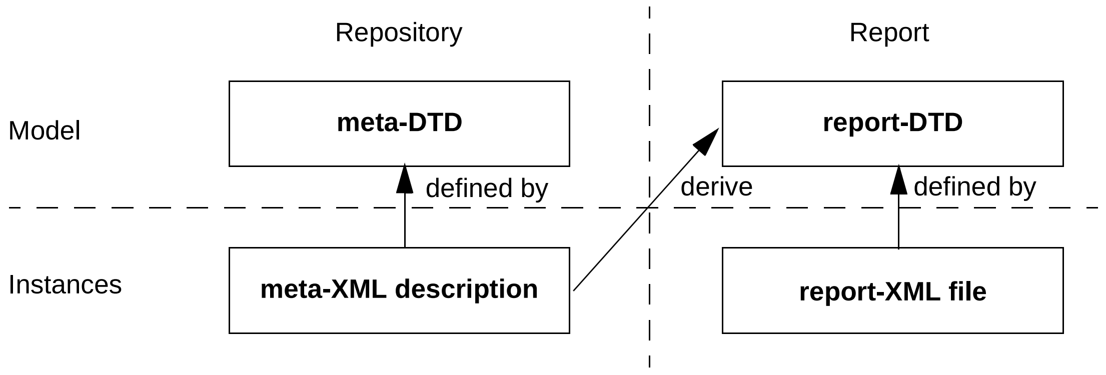


Figure 1. Repository and report modelling in terms of DTDs and XML-files

meta-XML description

```
<?xml version="1.0"
standalone="no"?>
<!DOCTYPE COMPELEM SYSTEM
"gtc.dtd">
<!-- demo-gtc.xml (GENVIS) -->

<COMPELEM name="Run"
          mapping="OBJECT">
  <LEAFELEM name="Date"
            datatype="Date"/>
  <COMPELEM name="Report"
            card="N">
    <COMPELEM name="Controller">
      <LEAFELEM name="PersonID"
                datatype="Long"/>
      <LEAFELEM name="Name"
               datatype="String"/>
    </COMPELEM>
    <LEAFELEM name="NumOfCases"
              datatype="Double"/>
    <LEAFELEM name="Comment"
              datatype="String"
              opt="YES"/>
    <COMPELEM name="Case"
              card="N">
      <LEAFELEM name="Date"
                datatype="Date"/>
      <LEAFELEM name="Amount"
             datatype="Currency"/>
      <LEAFELEM name="Remark"
               datatype="String"/>
    </COMPELEM>
  </COMPELEM>
</COMPELEM>
```

report-XML file fragment

```
<Run>
 <Date>2000-02-01</Date>
 <Report>
  <Date>1999-09-06</Date>
   <Controller>
    <PersonID>234985</PersonID>
     <Name>Edmund Ihler</Name>
    </Controller>
    <NumOfCases>3</NumOfCases>
    <Case>
     <Date>1999-08-25</Date>
     <Amount>123000</Amount>
     <Remark>
       Keine Aktionen nötig
     </Remark>
    </Case>
 ...
 </Report>
</Run>
```

Figure 2. A simple meta-XML description of a report repository (left part) and a fragment of a corresponding report-XML file (right part)

Since reports are stored as XML documents in the repository, we need a corresponding document type definition called *report-DTD*. This report-DTD is derived automatically from the meta-XML description by our system. Each report of our business context has to be a valid XML document conforming to a report-DTD. The relationships between the different DTDs and XML files are sketched in Figure 1.

Figure 2 gives a simple example to demonstrate a meta-XML description and a corresponding report-XML file containing a report data set. We allow data type constraints to be defined for attributes in report data sets. Concrete constraints are defined in the meta-XML description. For example, the attribute *datatype* in the meta-XML description is a constraint specification for the corresponding elements in the report data set. In the sequel of this paper, we will use this simple repository specification to demonstrate the generation of data visualizers. Real life repository specifications as used in our report management system are, of course, much more complex.

2.2. Layout model

The requirements for data set visualization are customer-driven. They depend on corporate identity issues and on different organizational roles of end users which require different views on the report repository. Therefore, we have to provide a set of different data visualization for a given repository model. Each single data visualization has its own view on the repository and its own look-and-feel for the end user. This motivated us to define an abstract model for data set visualization and to provide flexible means to customize it. Our abstract model consists of a grid-based cell layout, i.e., the user interface is organized in a rectangular grid of *cells*. The layout model supports three mechanisms to organize and customize cells in that grid.

- *Cell types* are either empty cells, label cells or data cells. Data cells are place-holders for data content in the repository. They are either single data cells or list data cells. If a list data cell is inserted in the layout, this cell is replaced by a list of data sets from the repository. Furthermore, we support label cells and - as aggregate cells - left or top labelled data cells. The latter combine a data cell with a corresponding label cell. This is important, since labels usually depend on the repository model. All cell types may have variable row and column sizes, i.e., they can span multiple rows and/or columns in the rectangular grid of cells.
- *Cell styles* are the means to specify the formatting of cells. They define cell properties (colors, fonts, alignment, etc.). Cells of different types may share common styles and cells with different styles may have the same cell type.
- *Cell groups* are the third mechanism. They allow to attach a set of cells in the layout to a cell group. An important feature of cell groups is their ability to inherit cell styles. A cell group inherits all cell styles from its parent cell group but may override them. Cell groups are organized in a tree. This results in an inheritance tree of cell groups which is a suitable means to define and propagate default styles in a very flexible manner. Nevertheless, each cell may specify its own local cell styles which override all cell styles inherited from parent group cells.

2.3. Model mapping and visualizer generation

The task is now to define a mapping that allows to associate elements of the repository model to cells and cell groups of the layout model. We make the mapping as simple as possible in order to achieve flexibility. The following mappings are required and supported:

- single data fields are mapped to data cells or labelled data cells,
- list data fields are mapped to cell groups, i.e., the element fields contained in the list are mapped to the cells attached to the cell group, and
- aggregate data fields are mapped to cell groups.

The meta-XML description of the repository, the layout model specification and the mapping specification between both models are the first input to the code generator of GENVIS. The second input are template files containing code fragments in the target language required to construct a data visualizer. These template files contain all elements specific for the target language. This approach separates target language dependencies from the generator and renders GENVIS adaptable to nearly arbitrary textual output formats. At present, there exist templates for LiveLink, XSLT-processors and for CRA-oriented [5] GUI-meta information.

Figure 3 explains the principle by three typical template file fragments. A code block definition starts with `%codeblockname` and ends at the beginning of the next code block definition. The generator assembles these code blocks to a data visualizer. During generation, special tags `<%...%>` are replaced by actual values. These values stem from the repository model or the layout model, or they are code generator specific variables. We will not go into further details here.

OScript template file fragment

```
...
%ListElementAccess
   `ListItem.<%Var.CellProps.ListTagPathRelative%>`
%BeginIteration
   ;Assoc ListItem
   ;for ListItem in .fReport.<%Var.CellProps.ListTagPath%>
%EndIteration
   ;end
...
```

Corresponding piece of generated code

```
...
   ;Assoc ListItem
   ;for ListItem in .fReport.Report.Case
...
   `ListItem.Date`
...
   ;end
...
```

XSLT template file fragment

```
...
%ListElementAccess
  <xsl:value-of select="<%Var.CellProps.ListTagPathRelative%>" />
%BeginIteration
  <xsl:for-each select="Run/<%Var.CellProps.ListTagPathParent%>">
%EndIteration
  </xsl:for-each>
...
```

Corresponding piece of generated code

```
...
   <xsl:for-each select="Run/Report/Case">
...
   <xsl:value-of select="Date" />
...
   </xsl:for-each>
...
```

CRA-like template file fragment

```
...
%ListElementAccess
   <ListElementAccess
      ListTagPath="<%Var.CellProps.ListTagPath%>"
      TagName="<%Var.CellProps.TagName%>"
   />
%BeginIteration
   <Iteration iterator="<%Var.CellProps.ListTagPath%>">
...
```

Corresponding piece of generated code

```
...
   <Iteration iterator="Report/Case">
...
   <ListElementAccess
      ListTagPath="Report/Case"
      TagName="Date"
   />
...
   </Iteration>
...
```

Figure 3. GENVIS template file fragments for OScript, XSLT and CRA data visualizer code. The generated code examples are related to Figure 2.

3. Prototyping of GENVIS

The tool GENVIS was constructed using the rapid prototyping environment APICES [6]. APICES is tailored specifically in order to prototype design tools with graph-based graphical user interfaces in a very short time. Its overall architecture is depicted in Figure 4. The prototyping design steps required to construct GENVIS are explained in this section.

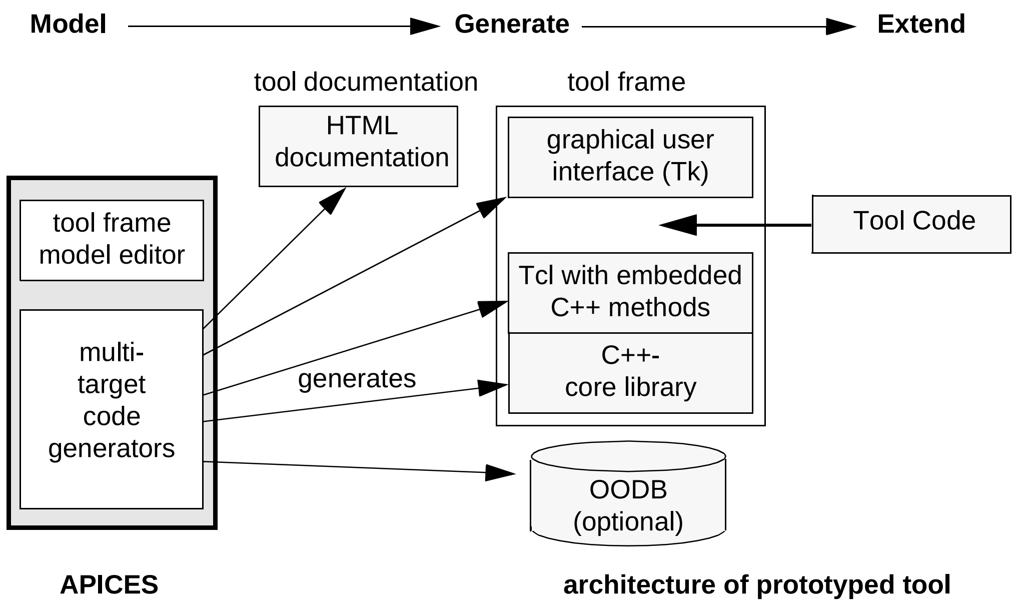


Figure 4. APICES and the architecture of a prototyped tool

3.1. Tool frame model

Prototyping with APICES starts with modeling the conceptual notions of the tool. In our case, GENVIS has to handle the model of the repository, the model of the grid-based cell layout and the model mapping between them. These notions and their associations are captured in a so-called *tool frame model*. The tool frame model is a UML class diagram [7] extended with a specification of the graphical user interface of the tool.

Figure 5 shows a screen shot of the tool frame model of GENVIS as it appears in the model editor of APICES. It shows the class diagram of the repository meta-model in the upper left part and the meta-model of the grid-based cell layout in the lower left part. The right part of Figure 5 shows an abstraction of those classes, whose instances will be visible later at the graphical user interface of GENVIS. In this context, the two boxes represent a left and a right region of the prototype's graphical user interface. The left one will display the repository model showing instances of the classes CompElem and LeafElem; the right region will show the layout model as an instance population of CompCells and LeafCells. Note that the class diagram of the tool frame model does not perfectly conform with the normative UML notation. In our tool, generalizations appear as dot symbols between classes with the arrow pointing to the more general class. Role names as well as optional inverse role names and their multiplicity are given as textual information in the middle of an association or composition link. The reason for not using normative UML is that the model editor itself is generated by APICES. An alternative would be to use a class diagram editor of an available

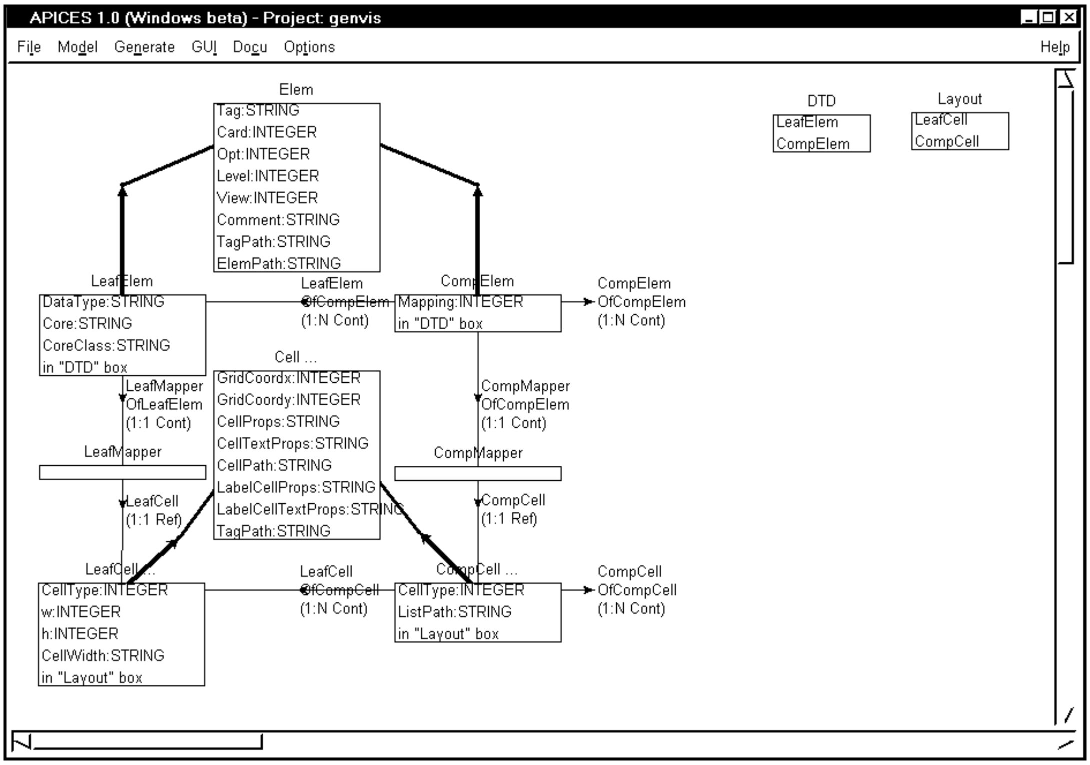


Figure 5. Screen shot of the APICES tool frame model for GENVIS

UML tool. We have already done work in this direction by implementing import and export filters to Rational Rose [8] for the class diagram part of our tool frame model.

3.2. Tool frames

APICES allows to generate so-called *tool frames* from the tool frame model. A tool frame is a refinement of the tool frame model to the following inter-related implementation targets.

- The first target is an object-oriented implementation of the UML class diagram in C++. It includes a full implementation of classes and all their associations. The generated C++ code is packed in a so-called core library.
- The second target is to embed the generated core library into the scripting language Tcl [9].
- The third target is a graphical user interface. We implement it by use of a generic GUI component, which is customized by parameters of the graphical user interface specification. The parameterized GUI component allows to edit a population of object instances.

The tool frame integrates all three targets. It may be considered an object-oriented application framework [10] generated from the tool frame model. This framework has well-defined hot spots [11] for application-specific extensions. Since extensions are done by method hooking rather than by subclassing, we may look at our tool frame to be a black-box framework.

3.3. Tool frame extensions

Tool-specific code, that can not be generated automatically, has to be hooked as extension to the hot spots of the tool frame. Extensions are implemented in the extended Tcl language. This allows to exploit all advantages of scripting [12] during the prototyping process. Furthermore, it opens up the generated tool frame for interfacing to other external components or tools [13]. In order to give an impression of the functionality we implemented in GENVIS, we list the most important tool features here:

- graphical repository and layout model editor (cf. Figure 6)
- import and export of meta-XML descriptions (cf. Figure 2)
- export of report-DTDs
- default layout generation for a given repository model
- highlight mapping between repository model and layout specification
- layout alignment to the grid and cell collision checking
- template-based code generator to produce data visualizers (cf. Figure 3)
- preview generation to check look-and-feel of grid-based layout (cf. Figure 7)
- comparison of repository model and layout model to check mappings

The overall code size we wrote for this functionality was 2500 lines of Tcl code hooked into the generated tool frame. The total size of the template file (Figure 3) to customize GENVIS to generate a OScript data visualizer is only 30 lines. This code comprises all template building blocks which are required to generate the source code of a data visualizer in OScript.

Figure 6 shows the generated tool interface of GENVIS. It displays the repository model on the left hand side and a concrete grid-based cell layout on the right hand side. The repository model is the graphical representation of the meta-XML file listed in Figure 2. Composite elements of the repository model are white boxes linked in a tree-structure. They contain leaf elements and links to contained composite elements. In the right part of Figure 6, a simple cell layout spanning 4 rows and 3 columns is specified. The different cell types of the cells result in different shading - content cells are bright, label cells are dark. The circles represent cell groups. The lower cell group aggregates three cells to which the leaf elements of the composite element *Case* are mapped. The arrows illustrate the mappings which the user specifies by associating elements with cells. Figure 7 shows a screen shot produced by a generated data visualizer. In this example, the data visualizer produces code for a XSLT processor. Note that the list of data sets in the lower part is specified by the last row of the layout model in Figure 6.

4. Experiences

GENVIS was developed in close cooperation by SYSTOR AG and GMD who provides the prototyping environment. GENVIS is in productive use in a SYSTOR AG project. In this section, we elaborate on experiences with the integration tool and its making.

The graphical editor of GENVIS simplifies the specification of repository models significantly. The visual modeling makes repository structures much more transparent than editing textual meta-XML files as done before. The meta-XML description of our system is also used to generate other components of our report management system, for example, SQL database schemas, data import filters, or XML parser implementations in C++, Tcl, or Java. Since we concentrate on the data visualization aspect here, a detailed description of these components is out of the scope of this paper. Nevertheless, GENVIS evolves from an integration tool for data visualization to the central specification tool for component integration in our project.

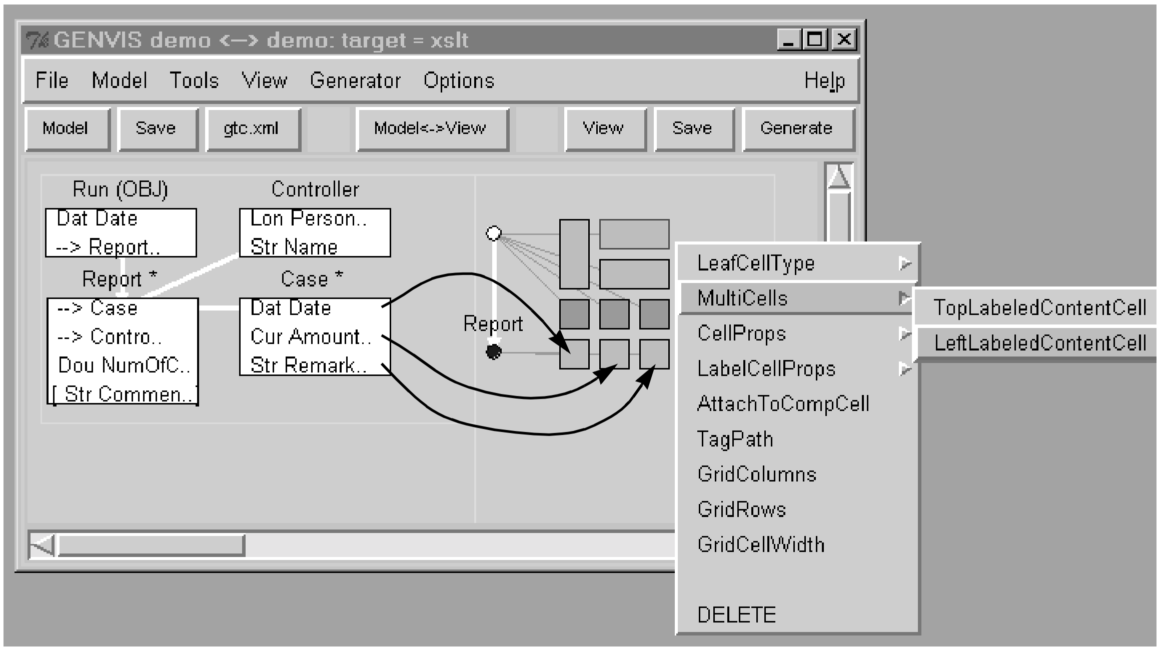
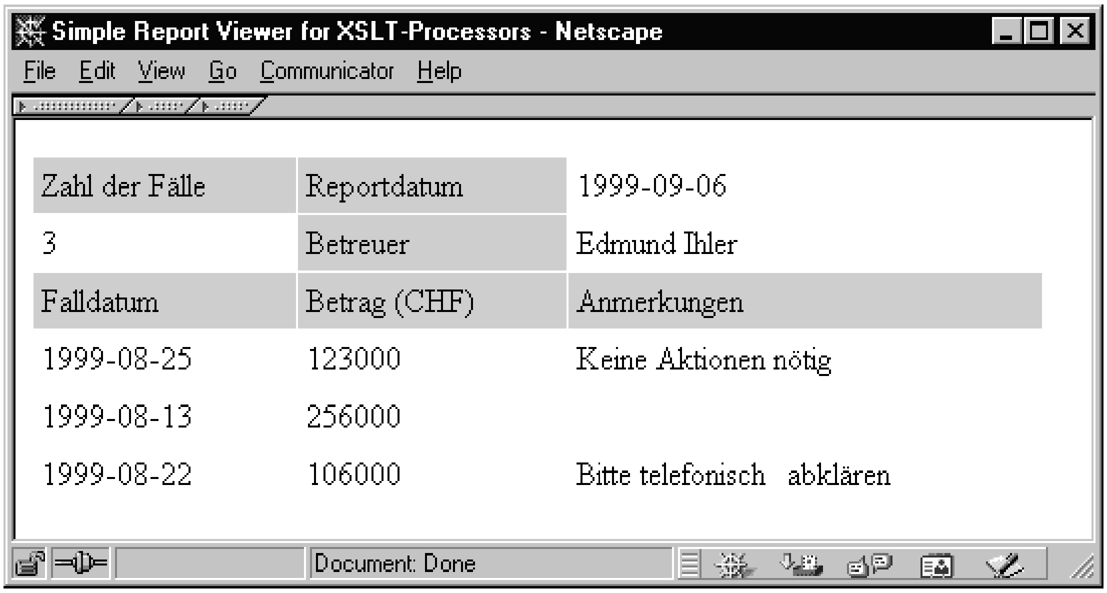


Figure 6. GENVIS user interface with a repository model in the left and a cell layout in the right. All screen objects have context-sensitive menus like the one displayed for a LeafCell object.


Simple Report Viewer for XSLT-Processors - Netscape
File Edit View Go Communicator Help
Document: Done

Zahl der Fälle	Reportdatum	1999-09-06
3	Betreuer	Edmund Ihler
Falldatum	Betrag (CHF)	Anmerkungen
1999-08-25	123000	Keine Aktionen nötig
1999-08-13	256000	
1999-08-22	106000	Bitte telefonisch abklären

Figure 7. Simple default report visualization for XSLT processor specified by the cell layout given in the right part of Figure 6.

The design of a specific user interface for the report management system is reduced to creating a cell layout by simply moving cells around and by specifying visual properties of cell groups and cells. Here, the property inheritance feature of cell groups turned out to be very helpful to perform complex but consistent style modification in the end user interfaces.

GENVIS is used in the production of report visualizers for the document management system LiveLink. Since the code generator of GENVIS strictly isolates target language dependencies from generic tool functionality, we are able to apply the code generator to other visualization targets very easily by adding new template files. At present, XSLT [3] and CRA-oriented GUI-meta-definitions [5] are additional supported targets.

5. Related Work

With respect to XML standardisation activities, XSL [14] and XML-Schema [15] are related to our work. XSL enables formatting information to be associated with elements in a XML repository. Thereby it allows the production of formatted output. In contrast to this textual mapping specification, GENVIS allows to sketch the layout graphically by arranging cells in a grid-based layout. The association of data elements to this layout is done by simply linking elements of the repository model to cells of the layout model. A further difference is that our generative approach is not restricted to generate data visualizers based on XSL. Other targets than XSL such as, for example, OScript code and CRA-oriented meta-definitions are alternatives for which we generate data visualizers. XML-Schema [15] is comparable to our meta-XML descriptions, but was not available when we specified our meta-DTD in 1998.

In the light of design methodologies using components like, for example, from Catalysis [16] or Rational [17], the prototyping tool APICES may be considered as a component generator. Our components are object-oriented frameworks with a pre-defined multi-language implementation architecture. These frameworks are a well-suited basis for stand-alone design tools [6] as well as for tools operating in tool environments [13]. In this paper, we demonstrated that our model-based generative prototyping approach is also well-suited to solve specific framework integration problems.

6. Conclusions

GENVIS solves our integration problem since it allows to generate source code of data visualizers for different visualization frameworks. The code generation is based on a mapping from a XML data repository model to an abstract grid-based cell layout. Specifying a data visualizer on the modeling level instead of implementing its code manually speeds up the design of our end user interfaces dramatically. Therefore, data visualizer generation with GENVIS increases our productivity. The separation of the layout model from the specific implementation of a data visualizer makes the tool reusable for the generation of nearly arbitrary visualizers for grid-based representation.

All in all, the benefits of GENVIS stem from two design principles. First, the tool operates on model level with respect to repository design and layout design. Second, the tool itself is based on a generated object-oriented application framework. The prototyping tool APICES can be used for a much broader range of applications, since a designer can capture the essentials of a prototype in a model and generate a dedicated framework for that model. The implementation architecture of these frameworks makes APICES particularly suitable for the family of design tools with interactive graph-based user interfaces (cf. [6][13][18]).

Interesting directions for future work on GENVIS are to generate Java source code for the SYSTOR AG framework JWeaver [4]. Another direction would be to generate data converters that are able to reformat XML documents so that they are suitable for mobile access via WAP browsers.

An interesting direction for future work on APICES is its integration into a commercial UML tool. First steps in this direction have already been done by extending APICES with class diagram import and export via Petal files [8]. Our experience with GENVIS encourages us to investigate the use of APICES in the problem domain of framework integration. Open research issues in this context are, for example, an extension of our framework generation approach to be capable of generating tool sets in a distributed environment.

References

[1] Mattsson, M., J. Bosch, M. E. Fayad, Framework Integration - Problems, Causes, Solutions, *Communications of the ACM* 42(10), 1999, pp. 81-87

[2] LiveLink, (http://www.opentext.com/livelink)

[3] Extensible Stylesheet Language Transformations (XSLT) Version 1.0, W3C Recommendation, November 16, 1999, (http://www.w3.org/TR/xslt)

[4] JWeaver, (http://www.systor.com/know/object/jweaver)

[5] The CRA-Cookbook, SYSTOR AG

[6] Bredenfeld, A., APICES - Rapid Application Development with Graph Pattern, *Proceedings of the 9th International Workshop on Rapid System Prototyping*, IEEE, Leuven, Belgien, June 3-5, 1998, pp. 25-30

[7] M. Fowler, UML distilled (2. edition), *Addison Wesley*, 1998

[8] Rose2000, (http://www.rational.com/rose)

[9] Ousterhout, J.K., Tcl and the Tk Toolkit. *Addison-Wesley*, Reading, MA, 1994

[10] Fayad, M. E., D. C. Schmidt, Object-Oriented Application Frameworks - Introduction, *Communications of the ACM* 40(10), 1997, pp. 32-38

[11] Pree, W., Design Patterns for Object-Oriented Software Development, *Addison-Wesley*, Reading, MA, 1994

[12] Ousterhout, J.K., Scripting: Higher-Level Programming for the 21st Century, *IEEE Computer 31(3),* March 1998, pp. 23-30

[13] Bredenfeld, A., Integration and Evolution of Model-Based Prototypes, *Proceedings of the 11th International Workshop on Rapid System Prototyping*, IEEE, Paris, France, June 21-23, 2000 (to appear)

[14] Extensible Stylesheet Language (XSL) Version 1.0, W3C Working Draft, March 27, 2000, (http://www.w3.org/TR/XSL)

[15] XML Schema Part 1: Structures, W3C Working Draft, February 25, 2000, (http://www.w3.org/TR/xmlschema-1)

[16] D’Souza, D., A. Wills, Objects, Components, and Frameworks with UML: The Catalysis Approach, *Addison-Wesley*, 1998

[17] Rational Software Corporation, Automating Component-Based Development, 1998

[18] Bredenfeld, A., Co-Design Tool Construction Using APICES, *Proceedings of the 7th International Workshop on Hardware/Software Codesign, ACM/IEEE*, May 1999, pp. 126-130