HotDoc
A Flexible Framework for Spatial Composition

Jürgen Buchner, Thomas Fehnl and Thomas Kunstmann
Darmstadt University of Technology
Programming Languages and Compilers
Alexanderstr. 10, 64283 Darmstadt, Germany
E-mail: {buchner,fehnl,kunstmann}@pu.informatik.th-darmstadt.de

Abstract

Building environments that use visual techniques (visual environments) is still expensive in labour. In this paper we present an object-oriented framework for compound documents, called HotDoc, that can be used for building visual environments rapidly. We give an example by describing a user interface for an extended spreadsheet environment called Simple. We focus on spatial composition control which is supported in HotDoc by layout policies.

1 Introduction

A number of visual environments exist that are built from scratch instead of using a framework. One reason for this might be, that the requirements for such a framework are not very clear. We have identified some general characteristics of visual environments that should be supported in such a framework.

- The user can place objects on the workspace. He or she can also delete, move and resize these objects.
- There can be an arbitrary number of objects and objects can have different types.
- Each object has a visual representation and also functionality (behavior). The functionality is accessible by activating the object (e. g. by double-clicking).
- The workspace can be saved, restored and printed.
- Relationships between objects are special objects and have to be visualized (e. g. by arrows). Dependant objects should be updated if objects are modified.

Using a framework that supports these general characteristics would allow rapid development of a visual environment and also support the reuse of components. Therefore we have built such a framework called HotDoc and used it for building visual environments.

2 HotDoc

HotDoc[1] [2], [5] is a framework supporting the implementation of compound document systems. A *compound document* is not simply a linear sequence of text, like the documents produced with the first generation of text processing systems. Instead, it is a hierarchy of *parts*. The information presented by the parts can be of different type, e. g. graphics, sounds, formulae or text.

HotDoc is an object-oriented framework and is implemented in VisualWorks Smalltalk.

Other systems supporting users in the construction of compound documents already exist. The best known are *Object Linking and Embedding* (OLE), *OpenDoc* and the *Andrews User Interface System* . We give a comparison between HotDoc and these systems in chapter 4.

2.1 Parts

A part of a compound document can be viewed as a combination of data and algorithms. The algorithms are used for presenting and editing the data. Therefore, it is a very natural way to model such a system as an *object-oriented system.*

In the HotDoc system, parts are real objects. A part consists of data and functionality to display this data and to interact with the user. HotDoc is like a "playground". The user can place parts on it and arrange them to his or her needs. Parts are like "windows" to applications.

[1]HotDoc is available at: http://www.pu.informatik.th-darmstadt.de/Projekte/HotDoc/

1049-2615/97 $10.00 © 1997 IEEE

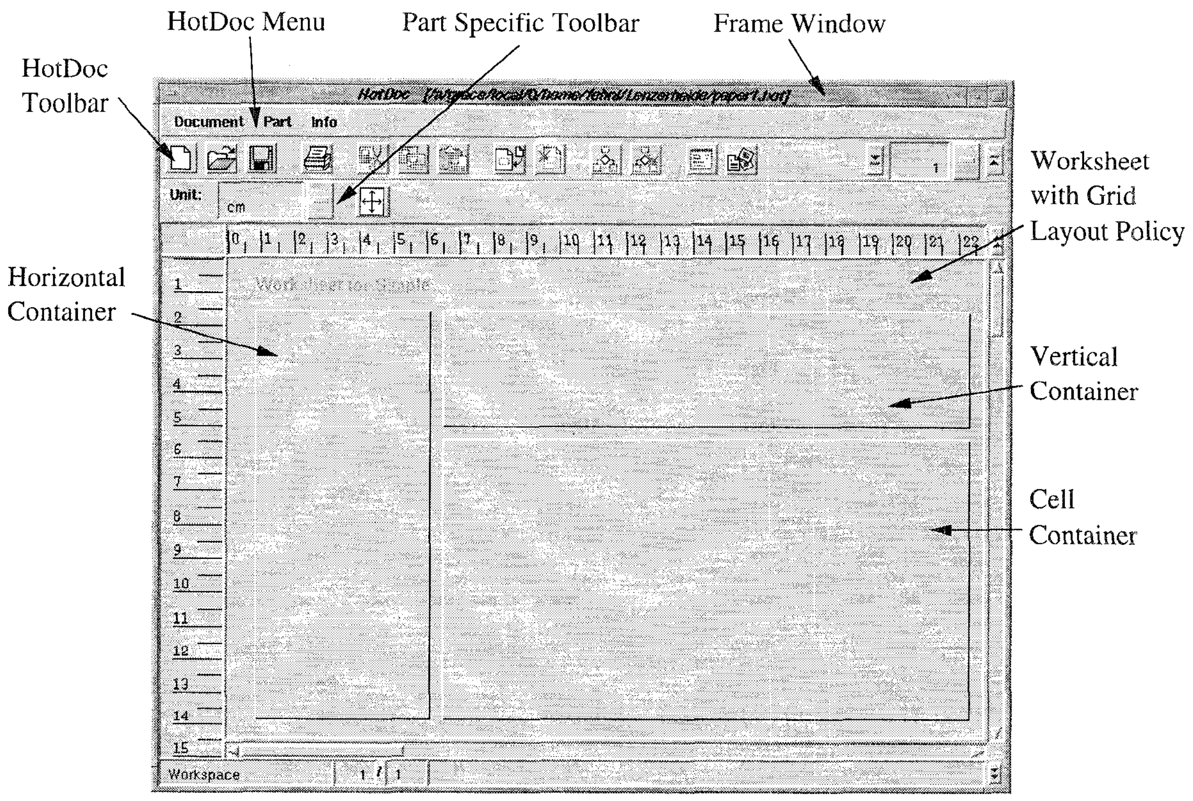

Figure 1: HotSimple Worksheet with three Containers

An empty HotDoc document consists of one special part, the so–called *workspace*. The user constructs the document by inserting parts into this workspace. These parts are called *children* of the workspace (see figure 1). Some children themselves accept parts for insertion. So, a HotDoc document is a tree of parts (see figure 2).

When a part is inserted, is is automatically activated. The activated part of a document is the part the user is currently working with. This part is marked with a thick grey frame. By clicking and dragging this frame, the part can be resized and repositioned. The user can activate another part by clicking on it (see figure 3).

The user interface of an active part is merged with the interface of the HotDoc system: The part's menus are included in the menu bar, the part's tool bar is displayed, the part's rulers are activated and the part's name is shown in the status line. This merging technique has the advantage, that the user interface basically stays the same. Additionally, written guidelines for the part designer achieve a uniform behavior. The user is not irritated by additional editor windows popping up.

2.2 Layout Policies

Placement of the parts is crucial. On one hand, the user should not need to place each part manually, on the other hand, a great flexibility of placement is necessary. Some kind of assistance to the user should be offered by the system.

HotDoc solves these requirements with so–called *layout policies*. A layout policy is assigned to a part and controls the placement of its children.

When the user moves a child with the mouse, the movement is constrained by the layout policy. After a movement, after the insertion of a new part and if the parent is resized, the layout policy rearranges all children if necessary.

In HotDoc some general layout policies are predefined. The designer of a part can decide, which layout policies are appropriate for the actual part. It is also possible to design special layout policies for specific parts. E. g. in a spreadsheet part it should be pos-

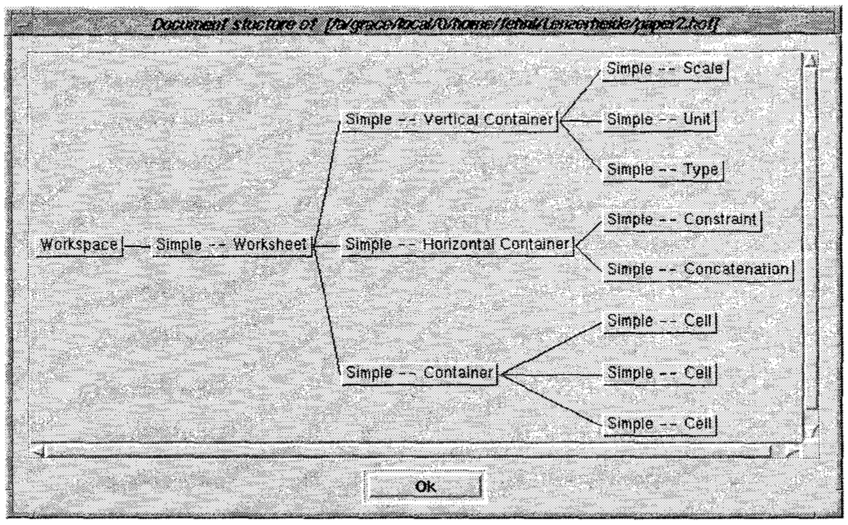

Figure 2: Document Structure

sible to insert parts in cells: If the size of the cell is changed, the size of the part should be changed too. The HotDoc spreadsheet solves this problem using a special layout policy. This layout policy constrains the positions of the children to the cell grid. If the cell grid is changed, the position of all children is recalculated. From the users point of view, the children parts are controlled by the spreadsheet. But the programmer only has to write a special layout policy, the rest is handled by the framework.

The framework is able to decide, which layout policies are appropriate for a part. The user can select or change the policy associated with a part at runtime.

To give the user more flexibility of placement, he or she can override the layout policy by pressing the control key while dragging the part. In this case, the part can be placed freely inside its parent. The part is not controlled by layout policy anymore. It is not included in rearrangement of parts.

2.3 Implementation of HotDoc

The implementation of the HotDoc framework consisted of three tasks:

- Implementation of a class library containing abstract and concrete classes. These classes are used by the HotDoc system itself and are also supporting the design of new parts.
- Implementation of the HotDoc frame window (figure 1). Through this window, the user interacts with HotDoc and the parts.
- Implementation of a standard part library. This is a set of standard parts which are useful for the construction of documents. This library includes a text editor, a bitmap viewer and some other parts.

HotDoc is implemented in VisualWorks Smalltalk [11] and uses the *Model–View–Controller paradigm* (MVC–paradigm, [6]) of Smalltalk. The idea behind this approach for the design of interactive applications is the partitioning in several objects: A (i) *model*, an object, which holds the data of the application, a (ii) *view*, which visualizes the model and a (iii) *controller* which controls the user interaction with mouse and keyboard.

Each HotDoc part is constructed according to the MVC–paradigm consisting of model, view and controller. Additionally, each part has an (iv) *application model.* This model is an extension of the VisualWorks system and links the user interface elements (menus, toolbar widgets etc.) with the part's view.

The fifth object for each new part is an (v) *I/O–handler* . It is responsible for persistency of the part. Because every document is a collection of nested parts, the whole document is persistent.

To implement a new part, the programmer can use and extend classes of the HotDoc framework. He or she can inherit from superclasses of the framework to define classes for all five objects of which a part is composed. It is not necessary to implement all classes for the part, but at least, a new view class must be

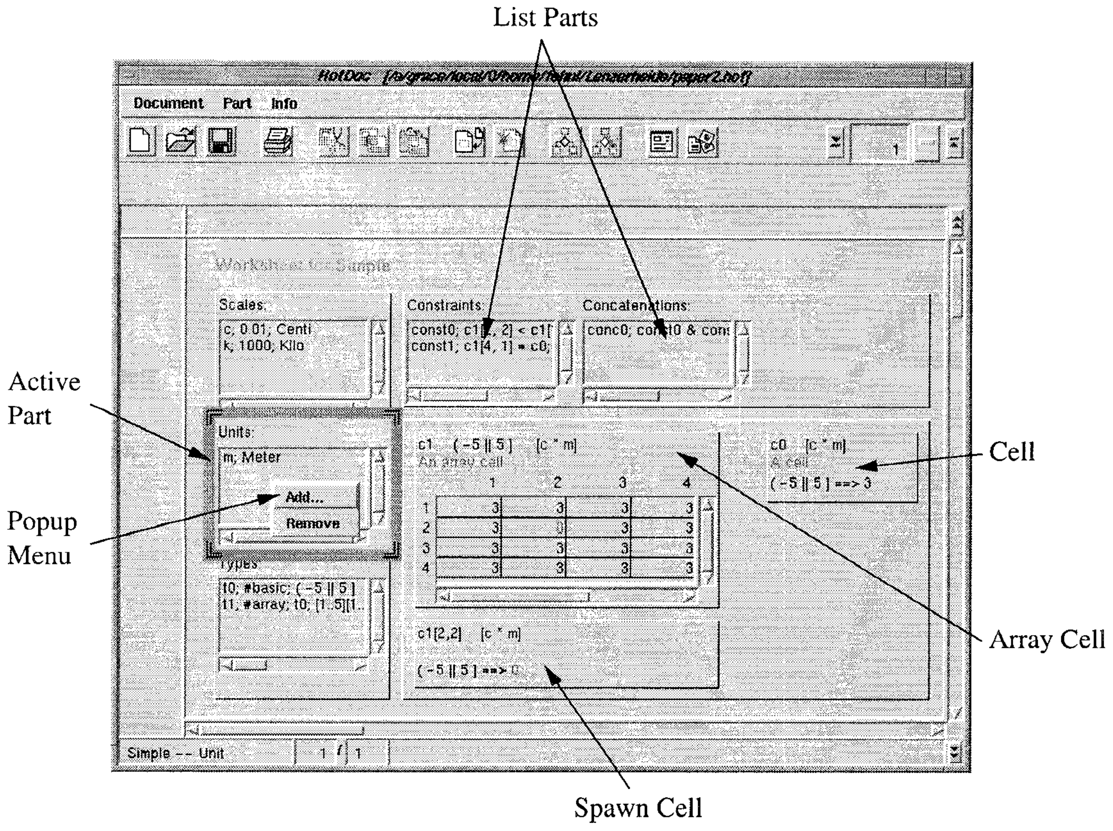

Figure 3: HotSimple User Interface

implemented. The view object is the first object created by the framework when a new part is inserted. Besides being responsible for the visual presentation of the part, the view object stores information needed by the framework to create all other objects for the part. Optionally the programmer may define a special layout policy for the part.

2.4 HotDoc for Visual Environments

Although HotDoc was originally developed for compound documents it can also be seen as a framework supporting the rapid development of a user interface for a visual environment. It supports all the characteristics identified in the introduction if the objects of the visual environment are modelled as parts in HotDoc.

3 Example

In this section we give an example of the creation of a visual environment by using HotDoc. We start by introducing Simple, an extended spreadsheet for simulation and planning. We then explain, how HotDoc was used to compose the user interface for Simple. The resulting visual environment is called HotSimple.

3.1 Simple

Spreadsheets are traditionally classified as visual environments (see [3]). But before focussing on the visual aspects of HotSimple, we first have to introduce the concepts behind Simple (see [7], [8]) which lead to special requirements for its user interface.

In traditional spreadsheets, the user describes a model and can later calculate certain results depending on the input he or she gives. In contrast the aim for developing Simple was, to support the user in exploring the model itself and in gaining a better understanding of the underlying processes. Therefore we added three characteristics that go beyond traditional spreadsheets:

- The description of a model in Simple is declarative, that is, it contains no direction of calculation. In other words, there is no distinction between input and output cells. This results in the ability to explore the model in an arbitrary direction (what–if and reverse–what–if, see [1]).

- Other than in traditional spreadsheets, the solution of a model in Simple is an n–dimensional space (where n is the number of variables in the model), which we call solution space. To explore this space, Simple provides a component, which allows the visualization of and navigation through this solution space. This enables the user to gain insight into dependencies between variables.

To support these characteristics Simple provides the following features:

- To avoid the direction of calculation we use multi–way constraints to describe a relationship between cells instead of formulas. A constraint is not bound to a certain cell, but is an entity of the model itself. Therefore it is possible to have multiple constraints for each cell. Constraints can be equations or inequalities that reference an arbitrary number of cells.

- Variables in Simple are cells whose value is not fixed but has to be in a given interval of values. Each variable has a current value which fulfills all given constraints. The current value of all variables can be seen as the current solution which is a point inside the solution space. Navigating in the solution space then means changing the current value of variables.

- Calculation of the solution space is done numerically instead of by symbolic computation. In a first step the solution space is calculated for each given constraint to allow the user to visualize parts of his or her model. In a second step the user can visualize the solution space for a boolean combination of constraints. It is therefore possible to experience the impact each constraint has on the final solution space. This is extremely valuable when dealing with over–constrained systems.

Another aim for the development of Simple was to to have more possibilities for structuring the models. Therefore we added two characteristics:

- The user interface supports the modularization of models.

- Models in Simple contain the possibility to check consistency and are more meaningful than in traditional spreadsheets.

Therefore we implemented the following features in Simple:

- The layout of cells on the user interface is not restricted to the usual grid. Instead cells (or groups of cells) can be placed anywhere on a worksheet. Cells can be combined to sets of cells. Constraints are possible between cells and also between sets of cells.

- To support consistency we introduced physical units in the models. This means that each cell is of a certain type which can be checked for type consistency like the data type in ordinary programming languages.

- The basic entities that make up a model in Simple are cells, constraints and the boolean combination of these constraints. Separating these entities enables the user to keep his or her focus on a certain aspect of the model he or she wants to describe.

3.2 HotSimple

We used the HotDoc framework to compose the user interface for Simple, which we have called HotSimple.

3.2.1 Implementation of HotSimple

One of the main ideas of Simple is the free arrangement of the cells. The HotDoc framework supports the free arrangement of parts on the screen. Therefore HotDoc was a very good foundation for the development of HotSimple. In HotSimple cells are not simply realized as parts but also for every element of Simple (e. g. cells, units, constraints) exists a part (see figure 3). By assembling these parts a user can create his or her own user interface. This flexibility supports the concept of Simple, where objects can be grouped on the workspace according to their relationship.

The HotSimple system consists of only sixteen parts which make up the whole user interface. One of these parts is a special worksheet part which is responsible for the management of the planning data. All other HotSimple parts have to be children of this special worksheet. Using the HotDoc framework the aspect of the free cell–arrangement can be ignored. The programming is reduced to the behavior of every part.

Each part reflects a special aspect of the domain model. It is responsible for visualizing this aspect and changing the state of this aspect in the domain model.

In HotSimple the visualization is mostly done by list parts. To change the state of a part a simple mouse click pops up a menu providing the needed functionality (e. g. add, remove, edit).

To provide an easy usage of HotSimple most of the parts have the same template. Most HotSimple parts consist of a label and a list, the so-called list parts. Ten of the sixteen HotSimple parts are list parts. To reduce the cost of development there is an abstract class for list parts. This abstract class contains most of the program logic. The subclasses of this abstract class have only a few short methods. For example the specification of the label. List parts need only six methods with one line of code each.

The HotDoc framework assists the programmer to develop persistent parts. For all HotSimple parts there are only two classes, four methods and 26 lines of code to handle the persistence.

3.2.2 Layout Policies

In HotSimple there are three layout policies. One for vertical layout, one for horizontal layout and one grid layout. With the vertical or horizontal layout policy new parts are automatically arranged horizontally or vertically after the last inserted part. If the user wishes these layout policies can also provide width and height of a part. If a child part is moved to another place these layout policies rearrange the other children and the horizontal or vertical layout is still valid. These two layout policies allow the creation of user interfaces with non-overlapping parts. HotSimple also supports a layout policy to arrange parts in a grid. The user can change the size of the grid and decide whether the grid is shown or not.

To make the usage of thess layout policies as simple as possible HotSimple comes with three container parts. Each of these containers has one of the discussed layout policies as default. Because these containers can hold other parts the user can create a clean and tidy worksheet with little effort.

Figure 1 shows a HotSimple worksheet with three containers. It has a grid layout policy as default, so the user could rapidly create such an arrangement of the containers. In figure 3 the containers of figure 1 contain other HotSimple parts. The layout policies of the containers arrange their children automatically. Figure 2 shows the document structure of the document in figure 3.

4 Related Work

In this section, HotDoc is compared with three systems for compound documents,

- Microsoft Object Linking and Embedding (OLE),
- OpenDoc and
- the Andrews User Interface System.

Another comparison between OLE and OpenDoc can be found in [4].

4.1 Object Linking and Embedding

OLE was introduced in 1991 as supplement to the Microsoft Windows user interface system. It is a set of protocols and services to enable application programs to share document parts [9].

If for example the user wants to edit a bitmap that was imported into a word processor, OLE starts the bitmap editor and passes the bitmap data from the word processor to it. When the user finishes editing the bitmap, OLE transfers the bitmap data back to the word processor. The word processor stores the data in the text document, but does not interpret it, because it is provided in the bitmap editor's native data format.

What the word processor actually displays is a Windows Metafile-representation (WMF) of the bitmap data. The WMF picture is also provided by the bitmap editor. Because WMF is the standard graphics format in MS Windows, no Windows application has trouble displaying such a picture. Using OLE no dedicated import/export filters are needed because each application is responsible for converting its data to the commonly understood WMF representation.

OLE version 1 could not conceal the fact, that the user works with a different application when he or she double-clicks on the bitmap to edit it. A separate program window containing the bitmap was opened. This was changed in OLE version 2 by introducing *in-place-editing*: Now the imported data can be edited inside the program window of the importing application. In our example of a text containing an imported bitmap the bitmap editor would take control of the word processor's window and display its own controls. To the user, this approach seems to extend the word processor by the features of a bitmap editor. However the basic concept of OLE is still to enrich a document created in a particular application by importing pictures from other programs.

HotDoc and OLE are very different approaches for constructing compound documents. An OLE document is a set of *passive* data blocks. These blocks are suited for printing or electronic transport. In OLE, there is not a single starting point for the construction of a compound document like HotDoc's frame window and workspace. The OLE user must decide, which application he or she uses as the root for the new compound document. This choice determines the basic layout of the document and the dominating document type. E. g. if a text editor is used as root application, the document will mainly consist of text. Other parts may be embedded, but they will be treated like "special characters" within the text.

In contrast, the HotDoc workspace is a "playground" for the user. Arbitrary parts can be placed on it or nested in other parts. HotDoc parts are not static, they are active "programs". Every part is like a window to some application program. All HotDoc parts have an uniform user interface, since they are activated in–place.

The in–place–editing feature of OLE allows a similar merging of user interfaces as in HotDoc. But this feature must be implemented by the developer of an OLE application. Only a few applications support this feature. In HotDoc, in–place–editing is supported by the framework with no additional costs for the developer.

4.2 OpenDoc

OpenDoc [10] is a platform–independent architecture for the implementation of distributed and component–based applications. OpenDoc was developed with the intention of splitting a huge application into small modular parts which can be combined freely. E. g. the text–editor, the spell–checker and the layout function are such parts. OpenDoc defines a set of document types, for each of them a so–called *event suite* is defined. An event suite defines a set of operations which an editor for some document type must implement. That's the reason why it is possible to use different editors for one document type in OpenDoc. In contrast, in HotDoc the association between document type and editor is fixed.

An OpenDoc document is constructed inside a *document shell*, this is a window similar to HotDoc's frame window. The document shell consists of a menu bar, a status line and the workspace. There is no support for tool bars or rulers like in HotDoc. Like in HotDoc, all editing in OpenDoc is done in–place. Parts are activated by mouse click.

In OpenDoc the parent part is responsible for the placement of its children. HotDoc has a much greater flexibility due to its layout policies.

OpenDoc supports inter–part communication and macro programming. These features are not supported in the current version of HotDoc, but are planned for the future.

The implementation of OpenDoc is based upon the *Distributed System Object Model* (DSOM) from IBM. The interface of a part is specified in a special description language and is translated to C or C++. The parts are implemented in C or C++. The implementation of a HotDoc part, which is written in Smalltalk, is much more readable and compact.

4.3 Andrews User Interface System

The Andrews User Interface System (AUIS, [12]) is developed by Carnegie Mellon University and IBM since 1982. A central component of AUIS is the *ez* editor. This editor has a similar role like HotDoc's frame window and OpenDoc's document shell: It is possible to place parts — in AUIS they are called *insets* — of different document types inside ez. Ez is mainly a text editor. Insets are handled like big characters flowing with the text. The flexibility of placement for parts is very limited. Like in HotDoc and OpenDoc, parts are edited in–place.

Like in HotDoc, it is possible to have more then one part working on the same data model. Additionally, relationships between parts can be expressed by a special scripting language called *Ness*.

In our point of view, AUIS has some essential drawbacks concerning its user interface design. E. g. part boundaries are not displayed and it is not visible, which part is active. Often the user changes the active part unintentionally.

5 Future Work

Until now, HotDoc lacks support for "programming" dynamic documents by the end user. While a part designer can do nearly anything with HotDoc by writing the appropriate Smalltalk code for parts and layout policies, the end user can only put preprogrammed parts on his or her workspace.

To overcome this problem, *visual scripting language* will be included in future versions of HotDoc. This language will be based on a preprocessor, which translates it to Smalltalk. This generated Smalltalk code is then compiled with the standard Smalltalk compiler.

The user will be able to assign identifiers to parts. Each part will have an interface of attributes and events defined by the part designer. In a script, the user will combine parts by reading attributes and triggering events.

Cooperative HotDoc is the extension of HotDoc to a CSCW system. Group work will take place in a "universe" of HotDoc documents connected by links. The cooperation is based on parts. Each of the cooperating users can edit some parts. Before starting to edit a part, the user must lock it first. The other users, which are working with the same document at the same time are informed about this locking by a red frame around the part. A part can be locked only by one user. The same mechanism can then be used to compose cooperative user interfaces like HotSimple.

6 Conclusion

We have identified some general characteristics of visual environments, that should be supported when using a framework for implementation. We introduced an object-oriented framework called HotDoc that supports them. We used HotDoc for building a user interface for a visual environment called HotSimple. Our experience in using HotDoc was quite positive, that is, it extremely reduced the amount of labour for building this user interface.

References

[1] Doug Bell and Mike Parr
Spreadsheets: a research agenda
ACM Sigplan Notices Vol. 28, No. 9, Sept. 1993, pp. 26–28

[2] Jürgen Buchner and Martin Hauck
HotDoc Programmierhandbuch
TH Darmstadt, 1996, Available from:
`http://www.pu.informatik.th-darmstadt.de/Projekte/HotDoc/PGuide/Paper/Paper.html`

[3] Margaret M. Burnett and Marla J. Baker
A Classification System for Visual Programming Languages
Technical Report 93-60-14, Oregon State University, Revised June 1994

[4] Paul Harmon
Compound Documents
Object-Oriented Strategies, Vol. VI, 1996, pp. 1–16

[5] Martin Hauck
Entwurf und Implementierung einer Klassenbibliothek zum Realisieren zusammengesetzter Dokumentstrukturen
TH Darmstadt, Masters Thesis, 1996

[6] Glenn E. Krasner and Stephen Pope
A Cookbook for using the Model–View–Controller User Interface Paradigm in Smalltalk–80
Journal of Object-oriented Programming, 1989, Vol. 1, No. 3 pp. 26 ff.

[7] Thomas Kunstmann, Martin Frisch and Robert Müller
A Declarative Programming Environment Based on Constraints
Proceedings of the IEEE Symposium on Visual Languages, Sept. 1995, pp. 120–121

[8] Thomas Kunstmann and Robert Müller
A Constraint Based Language for Spreadsheets
Proceedings of the Second International Conference on Practical Application of Constraint Technology, April 1996, pp. 445–452

[9] Microsoft Corporation
Microsoft Backgrounder: Object Linking and Embedding 2.0
Microsoft Corporation, Nov. 1992

[10] Chris Nelson
OpenDoc and Its Architecture
IBM, 1996, Available from:
`http://www.software.hosting.ibm.com/clubopendoc/aixpert_aug95_opendoc.html`

[11] ParcPlace–Digitalk
VisualWorks Cookbook
ParcPlace–Digitalk, 1995

[12] Andrew Plotkin, Wilfred J. Hansen et al.
A User's Guide to the Andrew User Interface System
Carnegie Mellon University, 1993