14

CONSTRAINT-BASED DEFINITION OF APPLICATION-SPECIFIC GRAPHICS

Manfred Pöpping, Gerd Szwillus

{poepping,szwillus}@uni-paderborn.de
URL: http://www.uni-paderborn.de/fachbereich/AG/szwillus/poepping.html
Universität - GH Paderborn, Fachbereich Mathematik/Informatik
Postfach 1621, D-33095 Paderborn, Germany

KEY WORDS: UIMS, User Interface Development Systems, Graphical Constraints, Application-Specific Graphics, Direct Manipulation

ABSTRACT: The development of graphical user interfaces with application-specific appearance and behaviour is still a challenge not met by current tools. We present the OBJECTION environment, which allows the designer to create application-specific graphical elements and incorporate them into the user interface. Objects are specified declaratively by adapting established techniques from interface builders for the creation of standard graphical user interfaces, enhanced by the use of graphical constraints to define graphical layout rules. Additionally OBJECTION provides a rich set of built-in interaction techniques to be added to the application-specific graphical elements on the user's request. Within an integrated environment the designer can create and test the application-specific graphics in conjunction with the standard interaction elements without having to undergo an inefficient edit-compile-test cycle.

1 INTRODUCTION

Graphical user interfaces (*GUIs*) like the one shown in figure 1 can be divided into a standard and a non-standard part. Standard interaction elements such as buttons, scrollbars and textfields (*widgets*) form the **standard part** of the user interface. The interface designer combines widgets with the aid of an interface builder or a user interface management system (UIMS), modifies the widgets' attributes and specifies reaction routines (*callbacks*) to be activated when a predefined event occurs. The UIMS or the underlying toolkit (e.g. OSF/MOTIF) manages the redisplay and dispatch of events during runtime. Hence, sufficient support for the creation of the standard part by current tools, such as MOTIFATION [Pöpping 92], exists.

However, much less support is available for the **non-standard part** of a GUI. If the application data is to be presented and manipulated in a more application-specific form, e.g. in a specialized diagram, standard widgets will hardly be applicable. In systems like OSF/MOTIF, currently used in practical user interface development, the designer has to deal with technical issues, such as redisplay and dispatch of events, himself.

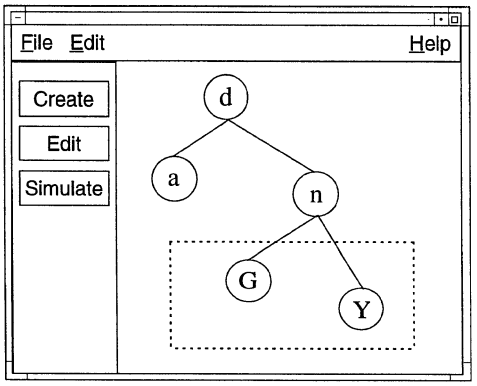

Figure 1: Sample GUI

K. Nordby et al. (eds.), *Human-Computer Interaction*

OBJECTION is a collection of tools supporting the interactive creation of new application specific interaction objects and thus aims at supporting the non-standard part of a GUI. The interface designer interactively creates

- the graphical **appearance** of application-specific objects.
- Graphical constraints can be used to describe the objects' graphical **appearance** and **layout** on the screen.
- The **interaction behaviour** can be specified by the selection of interaction properties and inclusion of procedural behaviour definitions, if necessary. The new application-specific object can be tested without leaving the integrated development environment.

This approach reduces the development time of application-specific elements and provides a declarative description layer for these, thus freeing the interface designer of laborious and error-prone hand-coding. During runtime the elements' redisplay and the dispatch of events is managed automatically by OBJECTION's runtime components. OBJECTION can be used in combination with MOTIFATION - an interface builder supporting the complete standard-part of a GUI based on OSF/MOTIF.

In section 2 application-specific graphical elements are described in more detail. Section 3 outlines the tool OBJECTION including its editors and runtime components.

2 APPLICATION-SPECIFIC GRAPHICAL ELEMENTS

Application-specific graphical elements (*ASGEs*) are used whenever standard widgets are not appropriate for representing the application data adequately. Examples can be found in various fields such as CASE, electronic design, CIM, or business planning. All sorts of symbolic graphical representations, such as statecharts, data-flow diagrams, module charts, wiring plans, layout diagrams, or task dependency graphs, are used for data entry and as representations in user interfaces for specific applications. Figure 1 is a very simple example of a diagram containing nodes (labeled circles) organized into a binary sorting-tree. We use this simple example to demonstrate OBJECTION's features comprehensively, and to respect the page limit of this paper at the same time although the system has already been applied to far more complex and realistic examples.

In the following we will discuss three major aspects of ASGEs specified with OBJECTION:

- **Data encapsulation** dealing with the inner structure of ASGEs,
- specification of the interaction behaviour of ASGEs under **direct manipulation**,
- and definition and evaluation of **graphical relations**.

2.1 Data Encapsulation

ASGEs are specified as object classes, implementing the principle of data encapsulation. The interface designer defines the internal structure by declaring attribute slots which contain application specific data. A dialogue box is created automatically, which is available both to the designer and the end user of the application for inspecting and modifying the data elements of an ASGE. Data may refer to the graphical appearance of an ASGE, such as its position, colour, or height, but may also be part of the semantic data of the application.

The creation of attribute slots automatically leads to the generation of access methods such as `set` and `get`. The interface designer can modify these methods by adding object-specific behaviour. Furthermore, specific methods can be added to define the individual behaviour of an ASGE. Methods are written in CODE, an interpretative version of the object-oriented language Objective-C.

It is possible to access application routines within methods. The designer can modify the underlying constraint set by adding or deleting constraints and the application can be unburdened from tasks such as consistency or plausibility checks. Furthermore, the designer can create and destroy objects, which the application does not be need to be aware of, because they are parts of the user interface only. This allows to relieve the application from having to deal with details of the user interface. An example for this is the line between two nodes in figure 1 which is only a representation element for depicting a relation established within the application, but does not correspond to a meaningful application object.

2.2 Direct Manipulation

ASGEs serve for the display and modification of application specific data. Every ASGE designed with OBJECTION can be selected, moved or resized without the designer having to write a single line of code. Furthermore it is possible to bind actions to these transformations which makes it very easy to implement direct manipulation features, specific to an ASGE. For example, drag & drop facilities can easily be implemented for an ASGE.

2.3 Graphical Relations

Apart from specifying single application-specific elements, the need to control their screen layout is apparent. Graphical constraints have been introduced by [Sutherland 63] and used in ThingLab II [Maloney 89] and Garnet [Myers 90] as a valuable concept for dealing efficiently with this problem. Their major advantage is their declarative nature: constraints are relations that only need to be specified once and are then maintained automatically by a constraint solver, one of the system's runtime components.

In figure 1, for example, the a-node is specified to be below the d-node, using a *below*-constraint, no matter where either one of the nodes is moved. A binary tree layout is automatically defined by the use of a combination of *below* and *leftOf* constraints. Another constraint applied here defines the nodes containing upper case letters to be situated *inside* the dotted rectangle.

Constraints can be added or deleted whenever an object is created or destroyed or an attribute value is modified. In general, the set of valid constraints can be modified during the execution of ASGE methods. Once the constraint is defined, the relation specified is kept valid automatically, without the designer having to program the corresponding modification operations by hand. The incorporation of the graphical constraint concept allows the declarative specification of graphical relations representing semantical relations between ASGEs.

2.4 Creating Elements

Once the ASGEs has been defined with OBJECTION the application programmer can use them in a similar way as standard interaction elements (*widgets*). The following commands demonstrate how to create and access ASGEs from within the application routines:

```
(1) node1 = [[Node alloc] init];
(2)  [node1 setValue: "d"];
(3) node2 = [[Node alloc] init];
(4)  [node2 setValue: "a"];
(5)  [node2 setLeftNode: node1];
```

In this example the a-node and the d-node of figure 1 are created (1, 3), the values of these nodes are defined to be "a" and "d" respectively (2, 4) using the `setValue` method, and the a-node is defined to be the left subnode of the d-node (5). The `setLeftNode` method call is associated with the creation of a line object and the installation of a constraint which connects this line object with the nodes and places the child node *below* the parent node. We show below (see figure 3) how this behaviour is defined as specific behaviour of the ASGEs of type `Node`.

3 THE OBJECTION ENVIRONMENT

OBJECTION provides several tools for the development of ASGEs. In the following we describe these tools while referring to the development of a very simple ASGE like the node in figure 1.

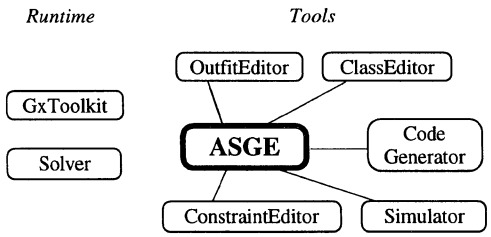

Figure 2: Components of OBJECTION

3.1 Creating ASGEs

The **OutfitEditor** is utilized to define the general appearance of an ASGE. The node class in figure 1 uses a simple circle with a centered letter as its outfit. The OutfitEditor allows the combination of predefined graphical primitives like lines, circles, labels or graphics. Even OSF/MOTIF widgets can be used in an outfit. Every outfit subelement can be

added to the list of attributes and can thus still be altered during runtime.
The **ClassEditor** allows the configuration of the internal structure of an ASGE, hence the definition and modification of attributes and methods.

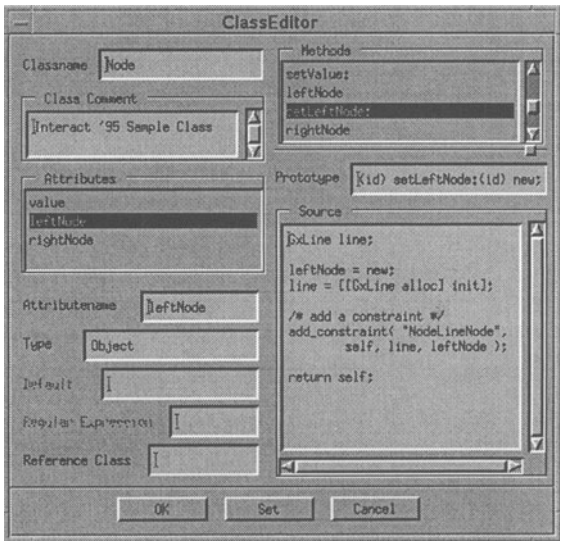

Figure 3: ClassEditor

Figure 3 shows a screenshot of the ClassEditor. The `Node` class currently developed owns three attributes and several methods. The source code of the `setLeftNode` method is displayed in the lower right text window. The CODE-statements in this window show that on invocation of the method a new line instance is created and the constraint `NodeLineNode` is applied to the new line. The interactive specification of the `NodeLineNode`-constraint is described in the next section.

OBJECTION's **Simulator** can be used to test the new interaction elements. The interface designer can create instances of the ASGEs under development, alter their attributes with the automatically provided attribute boxes and test the interaction behaviour, such as drag & drop facilities. The Simulator also allows the application of constraints, defined with the ConstraintEditor (see section 3.3), to ASGEs.

Additionally the Simulator permits to record and replay all creation and manipulation actions performed by the designer with an integrated recorder similar to a macro recorder. This enables the user to efficiently apply exactly the same manipulations to a set of ASGEs for debugging purposes.
Practice shows that during the development of an ASGE the designer frequently alternates between the ClassEditor and the Simulator. As both tools are simultaneously available within the OBJECTION environment, the designer is not forced to follow a strict Edit-Compile-Evaluate-cycle needed in a conventional setting.

3.2 Runtime Support

The following modules are part of OBJECTION's runtime system:

GxToolkit offers a set of graphical primitives like lines, rectangles and circles that are used to define the appearance of ASGEs. Even standard widgets are embedded to be used in ASGEs. Mechanism to perform selections, movements and resize operations are also part of the GxToolkit. This mechanism can be configured by many attributes. Either central callback functions for select, move and resize can be activated to maintain global control or a reaction for every single object can be defined. GxToolkit even adds facilities for grouping, ungrouping, loading and saving of objects. Every object based on GxToolkit is ready to communicate efficiently with the constraint solver.

The **Solver** is responsible for maintaining the constraints between the graphical objects. Objection uses the solver PARCON [Griebel 93] written by Peer Griebel. PARCON combines the algorithms unplanned firing, label inference and iterative methods to solve cyclic constraint networks consisting of arbitrary equations and inequalities combined by conjunction or disjunction. The solving speed is extremely high. A parallel version has been developed to gain even higher speed for large constraint systems. We prefer PARCON to the well-known DeltaBlue algorithm [Freeman-Benson 90], based on local propagation, because DeltaBlue is unable to solve cyclic constraint systems and is restricted to algebraic constraints. We are planning to incorporate another constraint solver, developed in our group [Szwillus 95] which uses a combination of linear and logical constraints to provide flexible support within user interfaces.

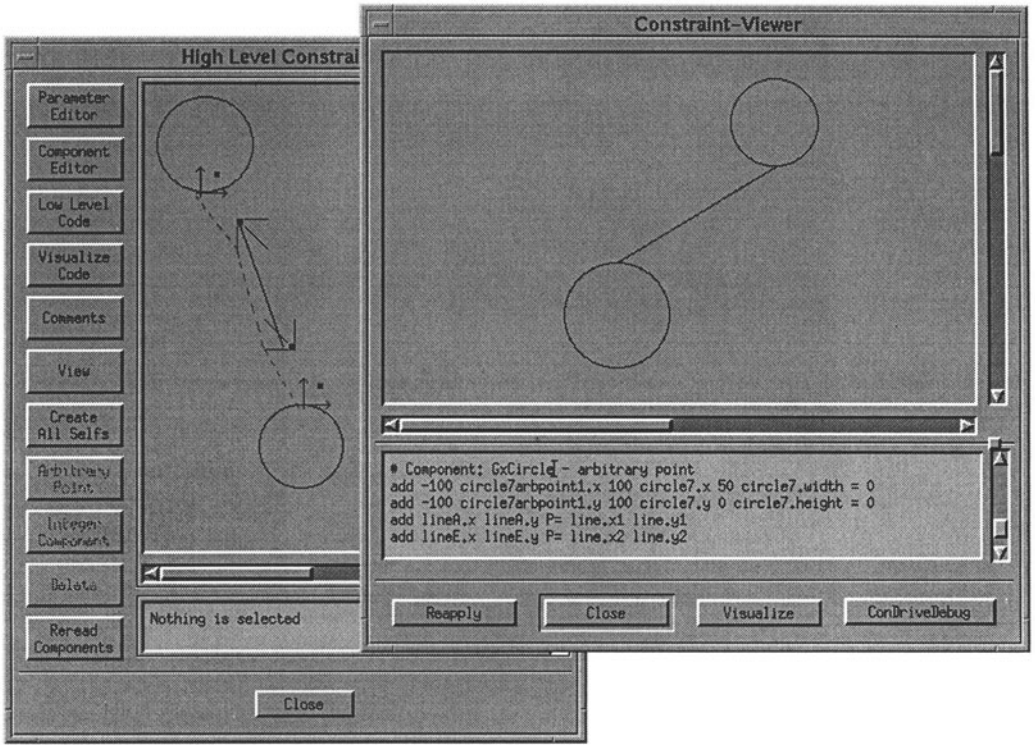

Figure 4: The ConstraintEditor

3.3 Editing Constraints

The **ConstraintEditor** allows the interactive development of new constraints. This can either be done by writing elementary constraints in the solver specific language or by combining previously defined constraints interactively. Writing base constraints requires a sound knowledge of the constraint specification technique because they must be expressed as a collection of mathematical formulas in the solver's input language. Many elementary constraints such as *pointEqual* or *pointOnLine* have already been specified and are provided for the user of OBJECTION. In most cases it will not be necessary to even look at this layer of specification.

For the interactive construction of a new constraint the types of objects participating in that constraint are specified first. In a pop-up menu the Constraint-Editor offers a list of available object **components**. Components are given depending on the object's type. The length or the starting point are just two examples for components that can be chosen when a line is selected. The central activity of the definition process consists of selecting object components and then applying previously defined constraints to them.

OBJECTION's ConstraintEditor enables the designer to create and immediately test new constraints. Figure 4 shows the design window on the left, where the `NodeLineNode` constraint is sketched and the view window on the right, where the constraint is tested.

The objects participating in the constraint depicted are two circles and a line. The first circle's component `BottomLineMidpoint` and the line's component `LineStart` have been chosen to apply a `PointEqual` constraint to them. The second `PointEqual` constraint has been defined between the `LineEnd` component and the `TopLineMid-point` component of the second circle. Finally a `PointBelow` constraint has been installed between the `LineStart` and the `LineEnd`

component of the line. The right window shows the three objects after the application of the constraint. The designer can select one of the objects, move them around and watch their behaviour, hence verify the new constraint definition*. Furthermore the ConstraintEditor allows to add a visualization for every constraint. This helps to find bugs, especially when a large number of constraints has been applied to objects.

3.4 Codegeneration

After the development of ASGEs, OBJECTION's codegenerator produces the appropriate source code for each class. In cooperation with the interface builder MOTIFATION, the source code for the complete user interface is generated, including the standard interface elements from the OSF/MOTIF widget set. The development is finished by inclusion of the final versions of the application functions.

In contrast to our system, other approaches such as Rockit [Kasenty 92] and Marquise [Myers 93] apply a different technique based on inferring methods to define constraints.

4 CONCLUSION

In comparison to conventional techniques, the development of GUIs can be simplified by using the OBJECTION development environment. To a large extent the designer uses declarative techniques for defining user interface elements. This applies to ASGEs properties as well as to their interconnections based on constraints. The development under MOTIFATION and OBJECTION covers almost the complete process of creating highly interactive, direct manipulation user interfaces.

The development of new constraints is a non-trivial design problem. There are cases where the resulting behaviour of a constraint does not meet the designer's expectations. We plan to implement a debugging tool that helps the interface designer to understand why a certain solution has been chosen by the constraint solver, and a more comfortable constraint visualization tool. Apart from minor improvements we want to apply OBJECTION to a set of realistic design problems to evaluate and improve the tool.

*) The bottom textfield displays the constraint in the solver specific language. This has been expanded automatically by OBJECTION for debugging only.

REFERENCES

Freeman-Benson, B; Maloney, J.; Borning A.: *An Incremental Constraint Solver*, Communications of the ACM, Vol. 33, No. 1, January 1990

Griebel, P.: *ParCon - a Parallel Constraint-Solver for the Support of Self-Organising Pictures*, (in German), 3. PASA Workshop, Bonn, April 1993, GI-ITG-Fachgruppe 3.1.2.

Kasenty S.; Landay J.; Weikart C.; *Inferring Graphical Constraints with Rockit*, Proceedings of the HCI Conference, Sept '92, pp 137-153

Maloney J.; Borning, A. Freeman-Benson, B.: *Constraint Technology for User-Interface Construction in Thinglab II*, OOPSLA '89 Proceedings, October 1-6, 1989

Myers, B.; Guise, A.; Dannenberg, R: *Garnet - Comprehensive Support for Graphical, Highly Interactive User Interfaces*, IEEE Computer, November 1990, pp 71-85

Myers, B.; McDaniel, R.; Kosbie D.S.: *Marquise: Creating Complete User Interfaces by Demonstration*, InterCHI '93, pp 293-300

Pöpping, M.; Griebel, P.; Szwillus, G.: *Motifation and Objection: Tools for the Interactive Development of User Interfaces*, (in German), GI-Fachgespräch "Innovative Programmiermethoden für graphische Systeme", Juni 1992, Bonn, Informatik, Fachberichte, Springer Verlag.

Sutherland, I.: *Sketchpad: A man-machine graphical communication system*, Proceedings of the Spring Joint Computer Conference, IFIPS, 1963, pp. 329-345

Szwillus, G.: *Solving Arbitrary Expressions of Graphical Constraints*, INTERACT '95