Modelling interaction tools in the Views architecture

E.D.G. Boeve

Computer Science/Department of Algorithmics and Architecture

CS-R9261 1992

Modelling Interaction Tools in the Views Architecture

Eddy Boeve

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Eddy.Boeve@cwi.nl

Abstract

Views is a user-interface system in which the user interface is a layer above applications, guaranteeing consistency of the interface, and with a data-layer implementing external object representation, allowing exchange of objects between applications without loss of structure. Although Views offers an architecture to deal with user interface aspects on a high level, in this paper it is shown that also low level interaction can be modelled with the architecture provided.

The
V I E W S
System

1991 Mathematics Subject Classification: 68U99.

1991 CR Categories: H.1.2, H.5.2, D39, H.5.0, I.7.2, D.3.3.

Keywords and Phrases: user interfaces, information interfaces and presentation, application environments, document preparation, constraint systems, window managers.

1 Introduction

As applications become more complex, the design of their user interfaces tends to become harder and harder. Brad Myers states in [8] that 60% of the effort of designing a specific user interface is put in the graphic design part of that interface. Therefore user interfaces for more complex applications are often created with so-called *user interface management systems* (UIMSs). More advanced UIMSs [3, 6, 7] are often provided themselves with a graphical user interface to manipulate the *interaction tools* (such as *widgets*) on the screen for the application to build: buttons, menus, sliders, text input boxes, etc. These interaction tools can be positioned and resized by mouse actions. Once positioned, other attributes can be edited to change colours, fonts and other properties. In most UIMSs the user interface created in this way is combined with the functionality of the application and linked with the routines for handling user input and system output for a specific window system to create the desired application.

This paper describes how basic user interaction in the Views System — as typing characters, clicking a button, resizing objects, etc.— can be modelled *within the Views system itself*. In fact, in the *designers* (application-builder) point of view there is no need for a UIMS or a high-level window system interface at all. In Views there is no distinction between the way the application appears on the screen and the actual functionality of the application: both are defined — whether interactive or not — within the system itself and doing this immediately results in a working application.

The first section gives a general overview of the Views System. There follows a description of the interaction model in Views with some examples of designing interfaces in Views. Finally, in the last section the conclusions are given and some remaining research topics are listed.

2 Views

Views [9] attempts to address user interface inconsistency by supplying a framework that new applications can be added to, guaranteeing a consistent and integrated user-interface across applications.

From the user's point of view, Views is a computing environment where all actions are achieved by editing documents, so that once you have learnt how to use the editor you can in principle work out how to do everything else.

From the application builder's point of view, Views is an open-architecture computing environment, where applications are easily added, the user-interface is guaranteed consistent across applications, and where user-interface issues are largely absent from applications[1].

Views has a data layer implementing external object representation, allowing exchange of objects between applications without loss of structure. Each object in the system has a *type*, which describes the internal structure of the object, and an *external presentation*, be that as text, as some picture, or even some other medium, such as sound. In general, objects can be viewed in different ways, even simultaneously, for instance as text in one view, but graphically in another. Structured objects are often called *documents* in Views. The presentation of a certain type on the screen is stored in the type's style sheet, named *presentation object* in the Views system.

1. Applications in Views are not applications in the normal sense. Because they all use the same user-interface, they tend to be more "specialised tasks" then applications.

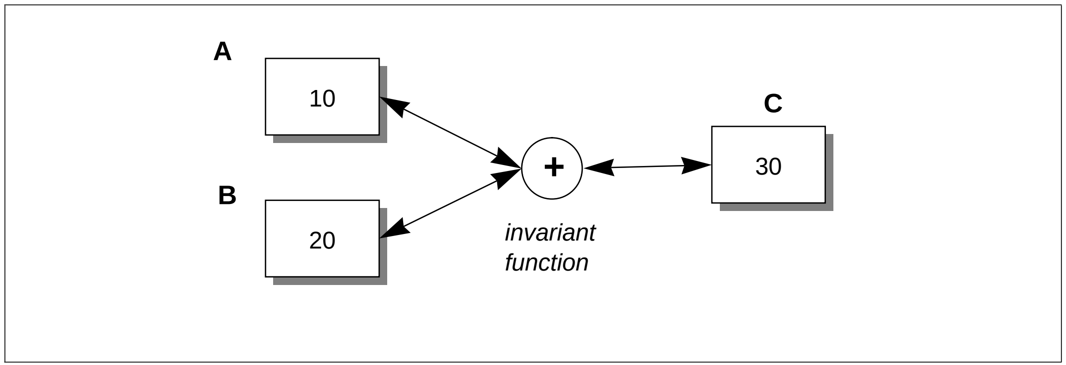

Figure 1: Three objects (A, B and C) connected with the invariant "A + B = C".

The main implementation model is that in general there are *invariants* (or *constraints*) between objects in the system. These invariants state that there is a direct relationship between the contents of an object and one or more other objects. If an object gets changed (usually by the user editing it), the invariant goes *out-of-date*, and has to be reinstated, which is done by calling a related function: the *invariant function*. See figure 1 for a diagram showing a simple invariant between three objects.

In contrast with other constraint-based user interface systems (e.g. [5]), Views has no explicit invariant *solver* or *satisfier* to decide which invariants should be satisfied, which rules should be used to satisfy each invariant and in what order the rules should be invoked to satisfy the invariants. Views uses *local propagation* to propagate changes through the invariant network and outdated invaraints are queued for recalculation. In principle, all invariants work two-way.

For all clearness, Views is not a UIMS, but is an environment in which the *designer* can develop and prototype new applications. The system will take care of the user interface layer and the data layer of the application under construction (see figure 2).

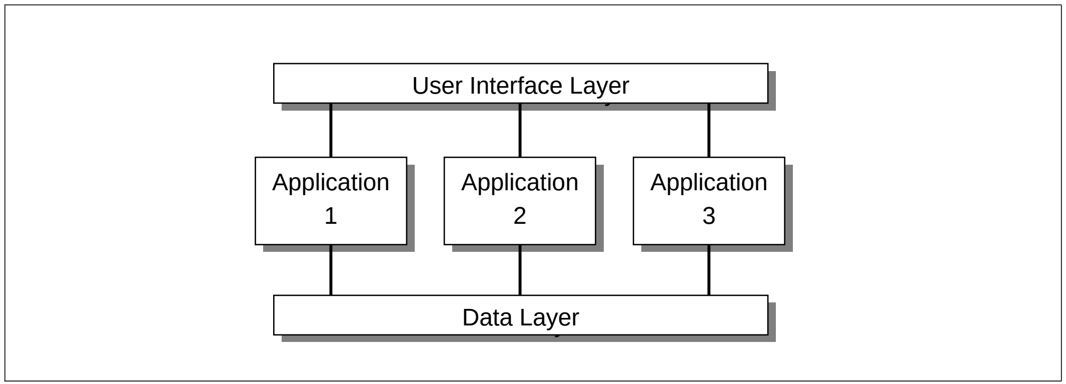

Figure 2: Applications under the Views system.

New applications can be easily added to the system and have a consistent user interface with respect to the other applications already in the system and the system itself. Because of the generic data layer it is easy to import objects defined by already existing applications into a newly created application.

In figure 3 we see a schematic presentation of the system. The dark shaded rectangle represents the plain Views System. User input events and screen output are handled by the so called *canvas layer.* This is a low level interaction level, implementing primitives for

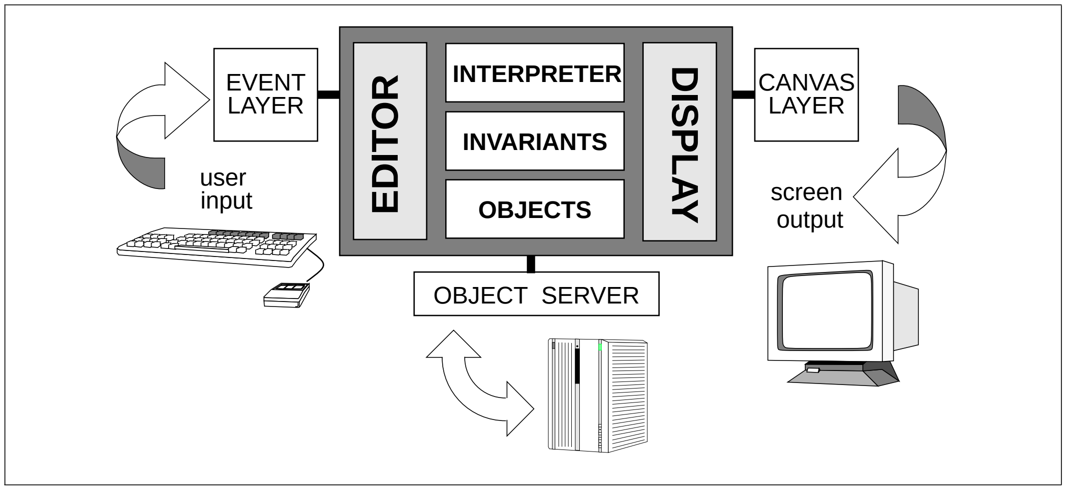

Figure 3: The Views system modules.

graphical screen manipulations and handling raw key and mouse input events. The high level interaction layer is implemented in Views, using a special type of interaction tool: the **interactor.**

3 Interaction in Views

The interaction tools found in other UIMSs can be stripped down in the Views system to more basic Views objects. One or more of these objects, possibly connected with invariant functions and fulfilling a specialised task (e.g. a *menu*, a *button* or a *slider*), is called an **interactor**. In figure 4 an example of such an interactor is given. The application this interactor belongs to has one internal object whose value can be toggled between the values one and zero by an OK-button.

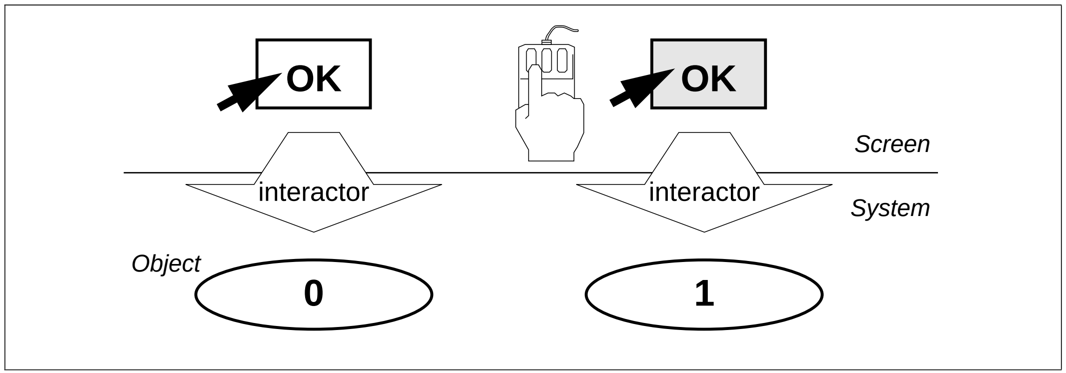

Figure 4: An example application with one internal object, whose value is controlled by a button.

An interactor can be seen as the interaction part of an application (see figure 5). Users can interact with the interactor by direct manipulation: by clicking, typing, dragging, etc. In general we distinguish three aspects:

- user interaction
- application interaction
- presentation

The user is interacting with the presentation of an interactor, i.e. the image Views presents the user on the screen, depending on an internal layout description of the object (in the object's *presentation document)*. The interactor converts events into commands for the application.

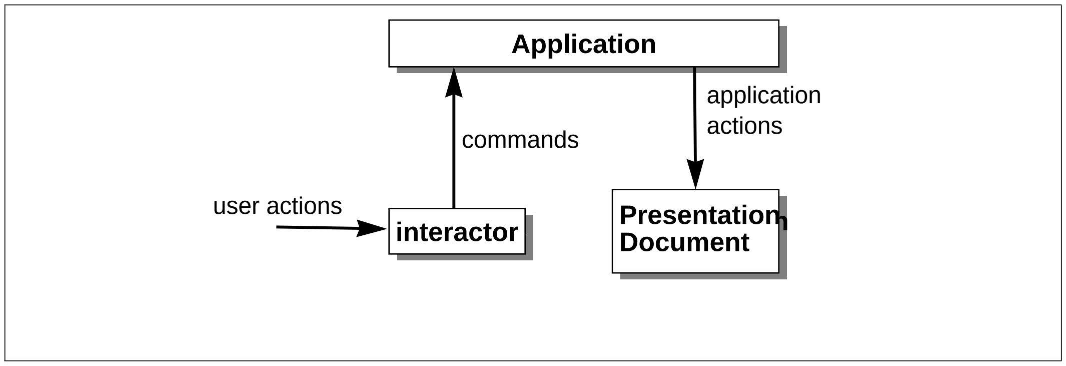

Figure 5: The flow of actions and commands between the interactor, the application and the user.

User interaction

When interacting with the system, the user will generate *events*. In general, events in the Views system are asynchronously generated by the user via *keyboard* and *mouse* input. An image of an event containing all relevant information of the event, is called an *event record*. Relevant information would be for example the key code of the character typed, the mouse pointer location or a modifier key code (*shift, alt*, *control*, etc.) at the moment the mouse button is depressed. Beside the user generated events, there are some system event sources in Views: the system timer for instance can generate *timer events* for timing purposes.

In the Views interaction model, interactive input is presented to the system by the canvas layer. It provides the system with a stream of mouse and keyboard events. There is one central **event record queue** in the system and entering an event record in this queue will trigger other actions. In figure 6 we see the event record queue and a invariant *Process Event Records* between the event record queue and the queues containing a subset of the incoming input events.

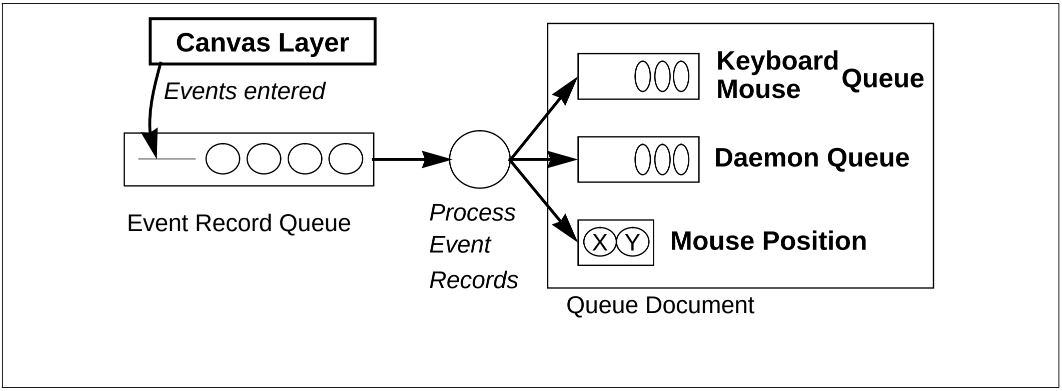

Figure 6: The main Event Record Queue.

The invariant function is a kind of *arbitrator*, filtering out the event records in this central queue into different objects in the **queue document**. For instance, the *mouse-move* event records are used to put the current mouse pointer position in the *Mouse*

Position object. But this is not enough: there are applications that are not only interested in the most recent mouse position, but also in the mouse *track.* They can not use this mouse position object because its value is overwritten during successive mouse movements. Therefore all mouse events are also entered by the arbitrator invariant in the *Mouse Queue*. Timer events are stored in the *Daemon Queue*. Applications that need regular impulses to work (like the Views clock) can link themselves to this queue.

The *Process Event Records* invariant divides the event queue into some sub-sets of its contents. Because memory is a limited resource, the items in the Event Record Queue are copied into one of the Queue Document queues and then *deleted* from the Event Record Queue. We still regard this as a invariant, although in the pure sense of the word, it is not.

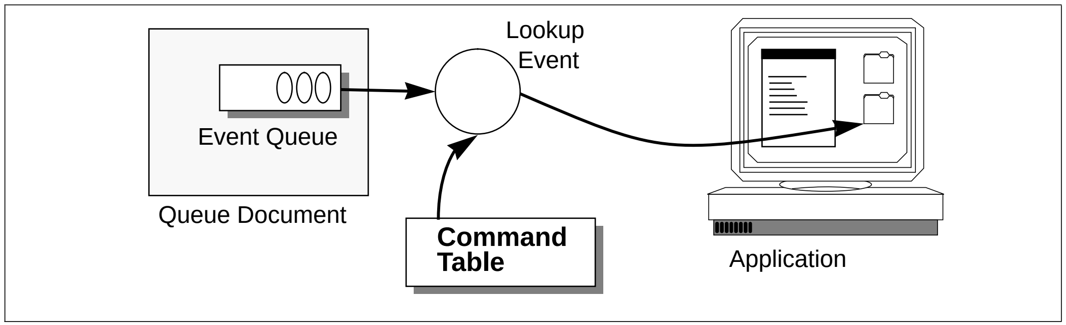

Figure 7: Event Handling.

In general, events stored in one of the queues in the Queue Document are converted to *commands* for the application by a *lookup* invariant (see figure 7). Events in the queue are looked up in the *Command Table* belonging to the application the queue is for. The Command Table contains “event — command” pairs and the command corresponding with the event is inserted in the Command Queue for the application.

To support multi-key codes (like the sequence of characters generated by an arrow key on certain keyboards) and modifier keys (like *shift* and *control* keys) certain events in the Command Table can be references to a sub-table, containing entries for possible completions for an event sequence.

In general, executing a command in the Command Table belonging to a user action is nothing more than triggering the invariants bound to the application.

Application interaction

The application can influence the behaviour of the interactor by changing fields in the object’s Presentation Document. It can, for instance, enable or disable a button or menu entry and change the interactor’s screen presentation.

Interactor presentation

Like other objects in Views, the presentation of an interactor depends on its Presentation Document. This document describes (amongst others):

- the shape of the interactor
- the interactor’s position on the screen
- choices for different presentations (an *enabled*, *disabled* or a *hidden* shape)

See [1] for an extensive description of the graphical aspects of objects in Views.

4 Using interactors

The designer using Views to build an application is offered a toolkit with high-level interactors, built from low-level interactors with standard presentation and interaction styles. The reason for this is *consistency*. By already filling in major parts of the Command Tables, interactors are always presented in a consistent way and they always react in a consistent way: the “look and feel” of an interactor is similar within one and across other applications within the system.

How can the designer use the interactors in the application she is building? There are two major ways of defining interactors in Views: with *direct manipulation* and by a *specification language*. The direct manipulation method is functionally the same as in other well-known UIMSs: interactors can be chosen from a toolkit, positioned somewhere in the application area on the screen, resized, etc. The values the interactors are controlling can be named by the designer and used in the applications she is creating. In the second method the designer uses the underlying specification language of an interactor.

In the next two paragraphs two simple, but complete applications under Views are shown to demonstrate this: converting degrees Celsius to Fahrenheit and vice-versa, and a bar graph application.

In the last two paragraphs examples are given of how to use the Queue Document.

The specification language presented in the examples shows only a possible form of the syntax of this language. The language has not been fully worked out yet and will probably change in notation and structure in the future.

Converting degrees Celsius to Fahrenheit

The application will appear to the user as in figure 8: she can either drag the thermometer leveller to the desired position or edit the value in the box.

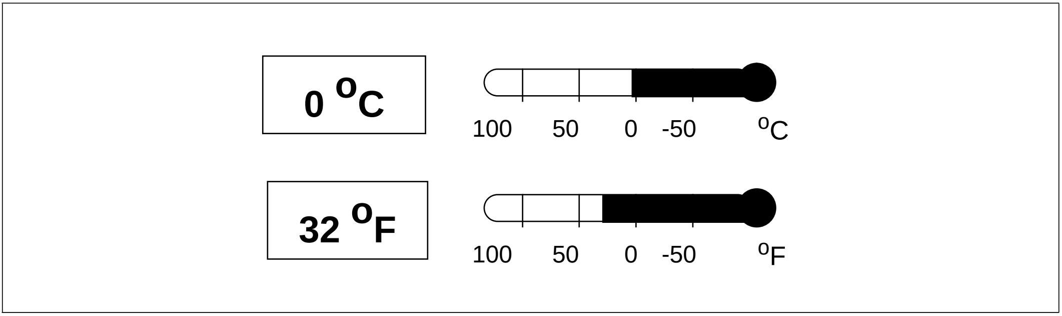

Figure 8: An application under Views: conversion from degrees Celsius to degrees Fahrenheit.

The application will show out to be quite simple, because in Views the functional behaviour of the application and its appearance on the screen are strictly separated. If we name the application `Temps`, the specification language could look like:

```
C_F_thermometer Temps

Temps.C = Convert( Temps.F )

function Convert( F ) = C
   where C = ( F - 32 ) * 5/9
```

Now we have to specify the presentation of the objects on the screen. We remark that both the numeric temperature value and the graphical value are in fact the same *object*, but with different *presentations*:

```
type C_F_thermometer( T ) = compound ( int C, int F )
   presented
      hang( Celsius( T.C ), Fahrenheit( T.F ) )

type Celsius( C ) = like int
   presented
      row( boxed( unit(C, “oC”) ), therm(C, “oC”) )

type Fahrenheit( F ) = like int
   presented
      row( boxed( unit(F, “oF”) ), therm(F, “oF”) )
```

In the code, keywords are underlined. `Hang` and `row` are functions that distribute objects given as parameters vertically and horizontally, `boxed` displays its parameters in a rectangle and `unit` and `therm` are pre-defined valuators to display continuous values respectively numerically and graphically.

Type information is specified with the “`<object> <type>`” construction. In more complex type definitions there will also be information about the presentation of the type when selected, disabled, etc.

A bar graph

In figure 9 we see a bar graph presentation of the sales figures of an item in four successive periods.

If we name the application `Sales`, the application language could look like:

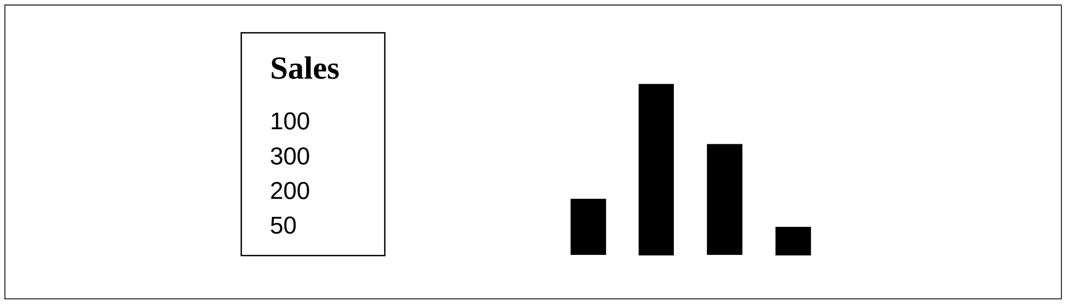

Figure 9: A spread sheet-like application under Views.

```
tablegraph Sales
Sales = { 100; 300; 200; 50 }

type tablegraph( list ) = sequence int
   presented
      row(     table( “Sales”, list ),
               bar( list ))
```

```
type table( T ) = compound (string title, sequence int list)
   presented
      boxed hang(T.title, hang(T.list))

type bar( list ) = sequence int
   presented
      width = 1
      row( filled( box( width, ∇ ) ) * list) )
```

As mentioned before, the specification language is a functionally oriented language. Operators found in other functional programming languages, like the `map` (`*`) and `reduce` (`/`) operators, are available (see also [10] and [2]).

The map operator applies a function to each element of a list of objects:

```
squared * [1, 2, 3, 4] = [1, 4, 9, 16]
```

Because `squared` is defined as a function that takes one number and returns the square of the number, we could write the latter also as:

```
squared(∇) * [1, 2, 3, 4]
   = [squared(1),squared(2),squared(3),squared(4)] =
   = [1, 4, 9, 16]
```

where ∇ takes subsequently one of the numbers of the list as input.

The `reduce` operator also takes a list of objects and a function and inserts the function between every elements of the list:

```
+ / [1, 2, 3, 4] = 1 + 2 + 3 + 4 = 10
```

The function `box` takes two parameters, width and height, and draws a box of that size. In our example, the height variable is taken from the number list. The `filled` function fills the graphical object it is applied to.

Mouse position

The invariant function processing the event records will update the *mouse position* object in the Queue Document at every mouse movement (as we saw in figure 6). This *global* position can be converted to a *local* position for an object on the screen with a simple invariant. For instance, in figure 10 the local mouse position object (X_1,Y_1) of a window is connected to the global mouse position (X,Y).

The mechanism for converting global positions to local ones is used so often in the Views system, that we will name the construction in figure 10 a *locator*.

Window queues

Events in a queue in the Queue Document have to be directed to the application they belong to. To simplify the distribution of events, the event stream is first split over the windows in the system. Once a window has been created, there is an event queue related with the window: the *window queue*. The events in the window queue are further split up into queues for the different applications in the window.

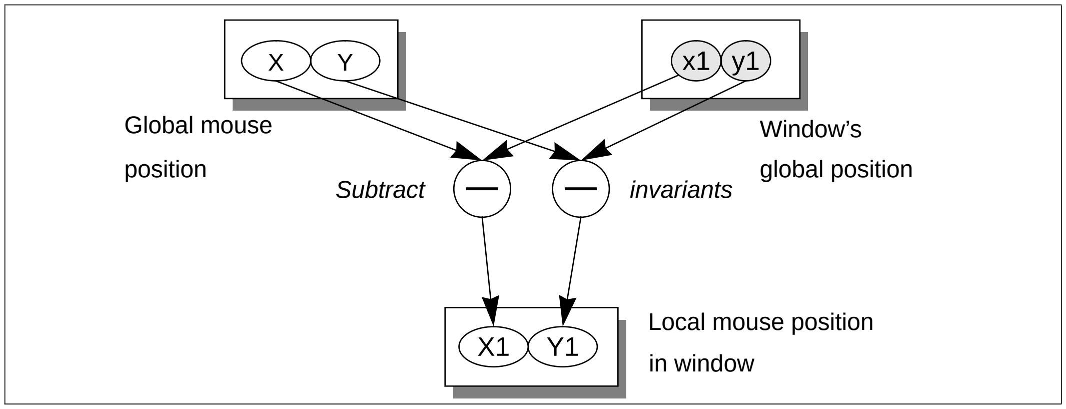

Figure 10: Converting a global mouse position to a local one.

One way to obtain the correct division between windows, is labelling event records with the current mouse position at the moment the event occurred. With help of a locator, the events can then be directed to the window where the mouse pointer is located. But this is not always correct: for instance, in a “click to type” window environment, there is a notion of a *current* window and even if the mouse is located outside the window, events are still generated for the active window.

To overcome this problem, we introduce *active* window queues. Events that cannot directly be related to a certain position on the screen (like keyboard events) are always directed to the queue of the *active* window. The window manager application can decide how a window can be made active, e.g. by single clicking in the title bar of the window or by entering the mouse into the window area.

Within one window, events are directed to the interactor they belong to (depending on the mouse position). As with windows, there will be an *active interactor queue* for events without mouse position information.

5 Conclusions and research topics

Designers of user interfaces will profit considerably of the Views system, where the design of a new application — both the functionality as well as the user interface — takes place on a high level, allowing a pure top-down design process. Although there are more user-interface design tools that allow top-down design of the user-interface within one unified tool (see for instance [4]), defining the applications within the system itself has big advantages above separate user interface management tools.

On a lower level, interactors in Views offers the designer a powerful method to build complex interaction tools. The designer of an user interface can choose one of the available interactors and, if necessary, combine them to more complex structures. Presentations and behaviour of interactors can be changed to fully agree with the desires of the designer and the user.

The mechanisms used by the interactors — objects and invariants — fits naturally in the Views approach. This will result in a system where every aspect of the interface can be manipulated in a consistent way and the interface with the out-side world (for instance the canvas layer and the keyboard) can be kept as small as possible.

The exact way the designer uses direct manipulation or the specification language to handle interactors has to be worked out more in detail. With regard to the specification language, we see the following research topics:

- How to design an easy to learn, but nevertheless powerful specification language?
- What should the primitives of the language be? (e.g. `pile`, `row`)
- Which library functions should be accessible to the designer? (e.g. `box`, `slider`, `button`)
- How can the specification language be translated into Views structures?

Direct manipulation asopects raises other research questions:

- How should the interaction model be defined? For instance, how should the actions followed by dragging a document icon onto the printer icon be modelled?
- Can the direct manipulation model be described in the same specification language as used for describing interactors and presentations?

6 Acknowledgements

Thanks to Steven Pemberton, Lambert Meertens, Lon Barfield, Guido van Rossum and Dirk Soede for their comments on earlier drafts of this article.

7 References

[1] Lon Barfield. *Graphics in the Views System.* CWI Report CS-R9260, CWI, Amsterdam, December 1992.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall International (UK) Ltd., 1988.

[3] Jan van den Bosch and Chris Laffra. *Project Digis: Building Interactive Applications by Direct Manipulation.* Computer Graphics Forum, vol. 9, pages 181—193.

[4] Andy Holyer. *Top-Down Object-Based User Interface Definition and Design Paradigms.* In Proceedings of East-West International Conference on Human-Computer Interaction '92, pages 421—428, August 1992, St.-Petersburg, Russia.

[5] John Maloney. Alan Borning and Bjorn Freeman-Benson. *Constraint Technology for User-Interface Construction in Thinglab II.* In Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pages 381—388, October 1989.

[6] Brad A. Meyers. *Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints.* ACM Transactions on Programming Languages and Systems, vol. 12, no. 2, April 1990, pages 143—177.

[7] Brad A. Meyers et al.. *Garnet — Comprehensive Support for Graphical, Highly Interactive User Interfaces.* Computer, November 1990.

[8] Brad A. Meyers. *User Interface Tools: Introduction and Survey.* IEEE Software, January 1989.

[9] Steven Pemberton. *Views: An Open-Architecture User-Interface System.* In Proceedings “Interacting with Computers: Preparing for the Nineties”, Noordwijkerhout, December 1990.

[10] David A. Turner, editor. *An Overview of Miranda.* Addison-Wesley Publishing Company, pages 1—16, 1990.