Programming aspects of Views. An open-architecture application environment

S. Pemberton

Computer Science/Department of Algorithmics and Architecture

CS-R9259 1992

Programming Aspects of Views

An Open-architecture Application Environment

Steven Pemberton

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

Abstract

Views is an open-architecture application environment that offers a consistent user interface across applications, interoperability between them, much less programming to produce an application, and the ability to add and modify applications on the fly. The system kernel contains a user interface layer and a persistent data-storage layer, so that applications only have to implement the true functionality. Objects contain no information on how they are to be displayed, so the presentation of documents can be changed easily, even on the fly. The main implementation technique is the use of invariants between objects which are automatically made two-way by the system. This document describes some aspects of programming for Views.

The
V I E W S
System

1991 Mathematics Subject Classification: 68Q52, 68N20, 68U99.

1991 CR Categories: D.2.6, D.3.3, D.3.4, D.3.2.

Keywords and Phrases: application environments, programming, invariants, programming environments, language constructs, language classifications.

1 Introduction

There are problems with current windowing environments.

The most obvious to users of such systems is the lack of consistency between applications. Even in environments renowned for their consistency (such as the Apple Macintosh) there are irritating and error-prone differences between applications, even at the system level: consider the Macintosh finder compared with the file-finder dialogue box. The task is the same (find a file to work with) but both the presentation (the look), and the manner of navigation in the filestore (the feel) are completely different (see Figure 1).

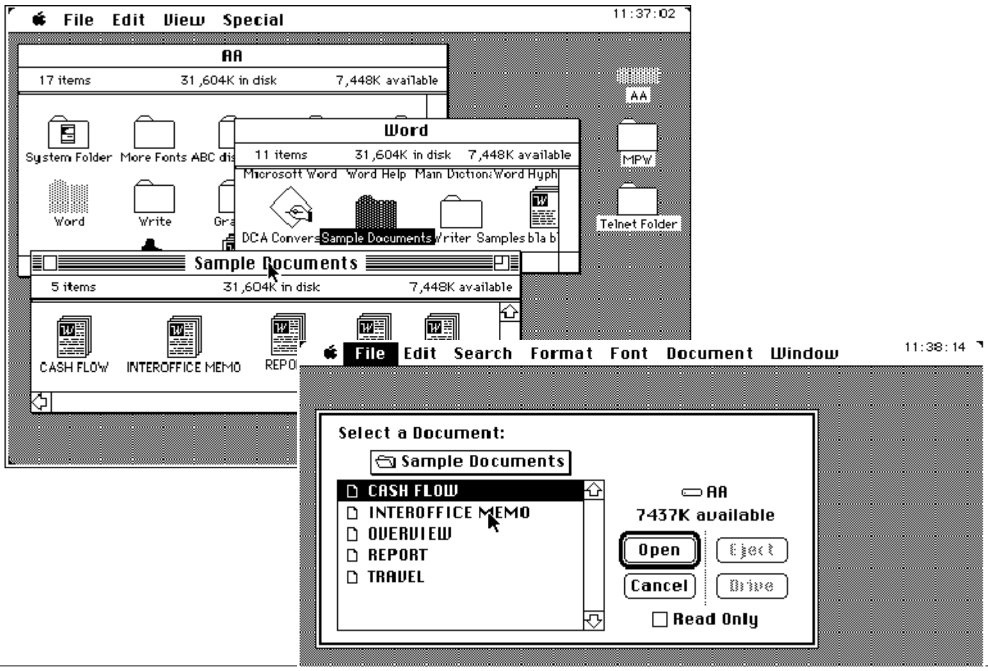
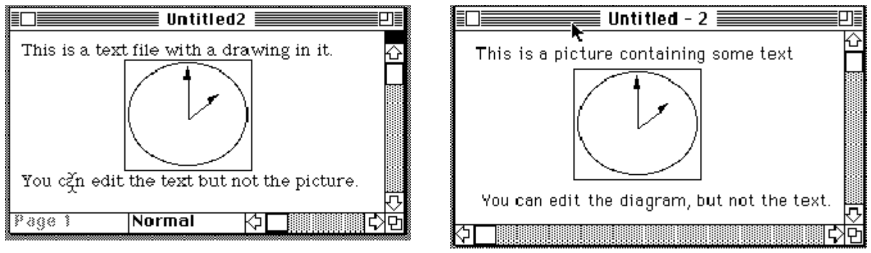


Figure 1. Within a short time, the new Macintosh user can be confronted with two different styles of browser, both with essentially the same task, but not recognisable as similar in appearance, and with different ways of navigating in the file-store

Another problem with current systems is a lack of integration between applications. Again drawing on the Apple Macintosh as a well-known example, it pioneered mixed-media applications, where for instance you can include text and pictures in a single document. But you must use different programs for producing pictures and producing text, so you have to decide whether you want a text with drawings in it, or a drawing with text in. They are both possible, and can produce identical results (Figure 2).



This is a text file with a drawing in it.

You can edit the text but not the picture.

This is a picture containing some text

You can edit the diagram, but not the text.

Figure 2. Two similar documents, produced with two different applications. If an application allows you to import documents from other applications at all, once you have done so, the structure of the original is usually lost.

Firstly then you must produce your base document, then you have to change context and run a different program to produce your drawing that you want to include in your text (or vice versa). Finally, once you have included the drawing, it is ‘frozen’: there is no chance to make further changes to it, except to delete it and replace it with a new version.

Another problem is the lack of interoperability between applications in windowing environments. In Unix-like environments, for example, many applications can be plugged together to produce new applications and tools. Even though the data-model used (single lines of text) is rather impoverished, a large range of applications can be used in this way. For instance, if you have a large, compressed, file of data, and you want to select the second field of each line that starts with the word ‘Total’, and total them up. It does not take much work to be able to write something like:

```
zcat data.Z | grep ^Total | field 2 | total
```

which would uncompress the file on the fly, select the lines, select the field, and cause them to be totalled. In windowing environments, similar modes of working are impossible; the best you can do is start each separate application up separately, and store intermediate results: it is generally not possible to link applications together to interoperate. This has caused a number of many-headed monolithic applications to emerge, allowing you to at least pass data between a small (fixed) number of applications, such as database, spread-sheet, and text-processor, or text-processor and picture editor.

While not all application environments have all these problems, they are typical examples of the sorts of shortcomings you can find in current environments. And all these problems come with another enormous difficulty: the cost of producing the user-interface. It is said that 90% or even more of the code of an application can be taken up with pure user-interface matters [1]. As an extreme example, the standard C “Hello World” program, 3 or 4 lines long, can expand to 250 lines or more if you produce a basic version for a windowing environment, but in general the work necessary to go from functionality to application is huge, and while the original “Hello world” will run on any computer that has a C compiler, you have to write a different version for each windowing environment there is, with hardly a line of code in common between the different versions.

These problems are caused by the isolation of applications from each other: they must each separately implement their own user interface (encouraging differences) and must each define and implement data-formats, discouraging integration and interoperability (see Figure 8.).

2 Views

Views [2] is an application environment that supplies a user interface layer as part of the system, and implements a data layer hiding all details of external data-formats and even the existence of disks and the like (see Figure 8.). This means that applications do not have to worry about user interface details, but are only responsible for the true functionality of the application. Since data formats and the user interface are handled by the system, any application may import objects from other applications, and those objects remain manipulable as if they were still in their original application.

This reduction in the areas of responsibility of an application, plus other factors to be discussed shortly, result in more than an order of magnitude reduction in the amount of programming needed for an application: for example, a clock program for a traditional windowing environment requires hundreds of lines of code. A similar application for Views needs a dozen lines, with the added benefits that the result can be used in other applications, the objects imported into other documents and so on.

Views is not a toolkit or a User Interface Management System, but an application environment. Emacs for example [3] is also such an environment, but with a rather simple, some would even say impoverished, data model (lines of text). Just as with Emacs, Views is not an environment that existing applications can be directly integrated into: applications have to be written for it; this is the only way Views can offer the degree of integration that it does.

3 The user interface

Since there is one single user interface supplied by the system, a model had to be found that would satisfy all potential applications. Preliminary research [6], [7] led us to what we have come to call the TAXATA model of user interaction: Things Are eXactly As they Appear. This has the following properties:

- All information is presented to the user as ‘documents’, in a broad sense (whether textual, graphical, or mixed).
- All objects are in principle editable
- All actions are achieved by editing
- The state of the screen exactly reflects the state of the ‘world’. This is a stronger form of WYSIWYG (What You See Is What You Get): as you edit objects the ‘world’ gets updated to match, on the fly.

An object can be for example a text document, the clock, a diagram, or a menu. Traditionally, editing a file is performed on a copy, which then has to be explicitly written back to the disk. The TAXATA model is closer to what one does in everyday life when changing a document. The one major advantage of editing a copy of a document – that you can retrieve the original version after a disastrous mistake – is offset by an unbounded undo mechanism in the Views data-layer.

As an example of the basic mechanism, consider document management. Instead of individual commands to list the documents that you have in a directory or folder, to rename them, to delete them, copy them, and so on, you just ‘visit’ the folder – which is a document in itself in Views – which causes its contents to be displayed on the screen. To rename a document in the folder you just edit its name; to delete it, you just delete its entry; to copy a document, you just use the normal copy and paste facility of the editor.

Similarly, to read electronic mail you just ‘visit’ your mail box. This causes its contents to be displayed as a list of message headers. Again, you can visit these individual messages (themselves documents), rename them, delete them, copy them, all in exactly the same way.

To print documents, you just copy them to the document representing the printer queue; to cancel printing, you just delete its entry; if you keep the printer queue document open on your screen, you can watch the progress of your printing jobs.

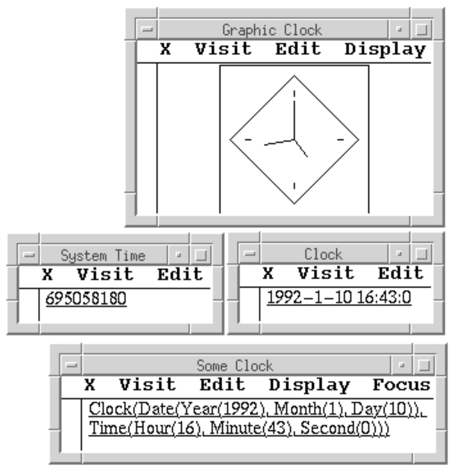
And so on for other tasks: reading news, listing and deleting running processes, editing textual documents, or amending a spread-sheet. Surprising applications, considered from a traditional point of view, include setting the time of day by editing the clock, and rearranging and renaming the menus, and redefining the shortcuts for menu entries, by editing a document describing the menus.

A major advantage that should be emphasised, is that once the user has learnt the basic actions of working with the editor, it should then always be obvious how to deal with a new application that the user hasn’t seen before.

4 Implementation

Views objects are structured, thus containing other objects, and consist only of ‘content-full’ parts: they contain no details of formatting or other display information, which is added separately when objects are displayed.

Each object has a type, which describes the internal structure of the object and its external representation, be that as text, as some graphical representation, 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 (see Figure 3).



695058180

1992-1-10 16:43:0

Clock(Date(Year(1992), Month(1), Day(10)), Time(Hour(16), Minute(43), Second(0)))

Figure 3. Four views of the time. On the left is the time as represented internally by the computer: the number of seconds since the beginning of 1970. This value is then projected in three different ways: as an analogue-style display of the time, as a digital-style display, and lastly showing the internal structure of the digital-style display, without any added formatting. In this last display, the type of each (sub-) object is given, followed by its value in brackets. Each display also shows its name, and has a ‘menu-bar’ for editing the object and controlling the display.

When an object is displayed, the description of its external representation is accessed and used to determine how the object should look.

Within the user interface layer there is a generic editor which knows about the structure of objects, and allows the user to edit all objects in the same way, regardless of how they are displayed.

In general there are ‘invariants’ between objects in the system (Figure 4).

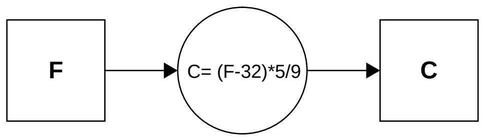


Figure 4. Invariants, here represented as a circle, link objects together. If an object is changed, the invariant is re-instated by changing linked objects. Here an object representing the temperature in degrees Fahrenheit is linked by an invariant to an object representing degrees Celsius. Since the invariants are two-way, if the user edits either one, the other gets updated automatically.

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 is re-instated by the system.

In general, the invariants are two-way, so that it doesn't matter which objects in an invariant get changed. Higher-level invariants, defined in terms of user-defined functions, get broken down into a network of low-level invariants by the system (see Figure 5).

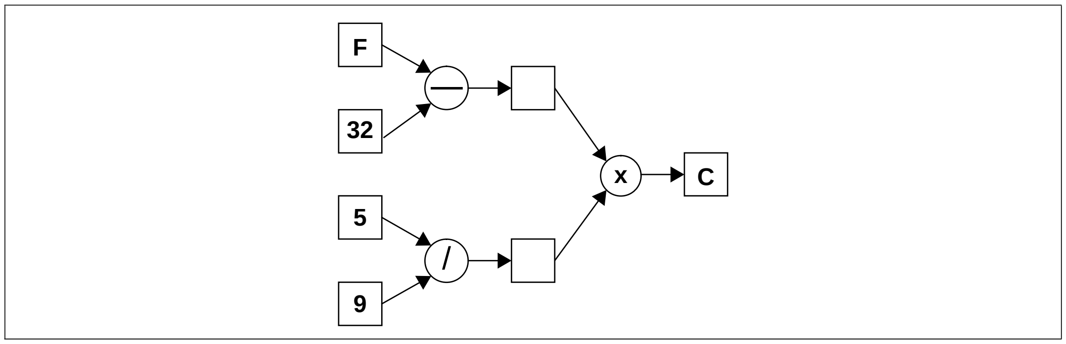


Figure 5. A high-level invariant, such as the one in Figure 4 is broken down by the system into a network of lower-level invariants

Local propagation [4] is then used to propagate changes through the network of invariants when any object gets changed; no knowledge of the structure of the invariant graph is used for constraint solving [5][1]. An optimisation is used that 'puts to sleep' invariants that are attached to currently unused objects (objects that are not visible on the screen and not part of a chain of invariants leading to a visible object).

In fact, the invariant mechanism is used very generally throughout the system, so that for instance, displaying objects on the screen is done by application of the invariant 'the representation on the screen must match the object'. When an object gets 'visited' (i.e. made visible on the screen), a window is opened, and a chain of invariants created that creates a presentation of the object, and causes that to be displayed on the screen. If the object gets edited, then the screen gets updated (see Figure 6 and Figure 7)

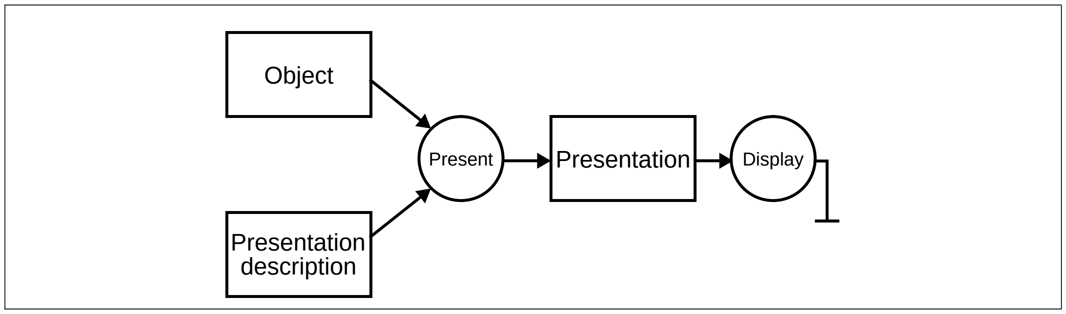


Figure 6. Presentation of documents on the screen is done using the invariant mechanism. When a document is 'visited' a new 'presentation' object is created, linked by an invariant to the object to be visited and the description of how the object should be displayed (itself an object); from the presentation object emerges a second invariant that maps the presentation onto the screen. If the visited object gets changed, the invariant goes briefly 'out of date' and is restored by the system, by updating the presentation document. Such an arrangement also means that if the description document gets changed (for instance, edited by the user) the presentation changes to match, on the fly.

.

1. In fact, there is a school of thought that says that such a system of equality constraints is not a constraint system at all. We bypass this problem by only referring to invariants, and not constraints.

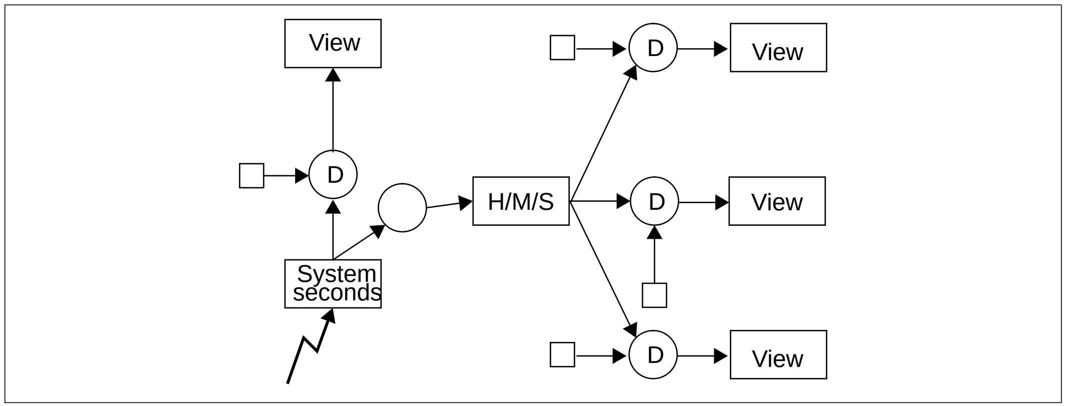


Figure 7. A simplified view of the internal connections of the objects displayed in Figure 7. The oblongs represent objects, and the circles invariants connecting them. If an object changes, any connected invariants are used to update connected objects. The system-time object gets changed automatically by the system each second. The diagram has been simplified to show the main relationships, but the display functions (marked with a D) for instance, are also connected to objects describing how the objects are to be displayed, and giving the details (size, etc.) of the windows in which they are to be displayed.

5 Adding new applications

To add a new application to Views, the application designer has available a large collection of built-in types and objects, and types and objects created for other applications. If necessary however, new objects can be defined: how they are structured internally, how they are to be displayed, and the relationships between the different objects in terms of invariants. The Views system takes care of the rest: input and changes to the objects, displaying objects, and ensuring that the invariants are kept up-to-date.

To add a new application then, the application programmer creates a new workspace, and follows a number of steps:

- defines any new data-types to be used, and their external representations
- creates the necessary objects (instantiations of the data-types)
- specifies the necessary invariants between the objects.

These steps are treated in more detail in the following sections.

6 Data types

Views supplies a data-layer to applications (see Figure 8). This layer hides all details of the external representation of objects; to an application all objects are directly accessible and the existence of disks and main-memory is transparent. The data-layer decides on the structure of objects on disk, and takes care of transferring objects from disk to main-memory as objects are called for and changed. All objects are effectively ‘persistent’: they only disappear when they are explicitly deleted. Even if you stop running Views, and come back later and restart it, the objects are still there.

Part of the data layer is an unbounded undo mechanism, so that all applications have automatically and transparently full undo available.

Apart from the pre-defined data-types available in the library (including primitive types such as numbers and strings), new data-types can be built up from a small, generally useful set of constructors:

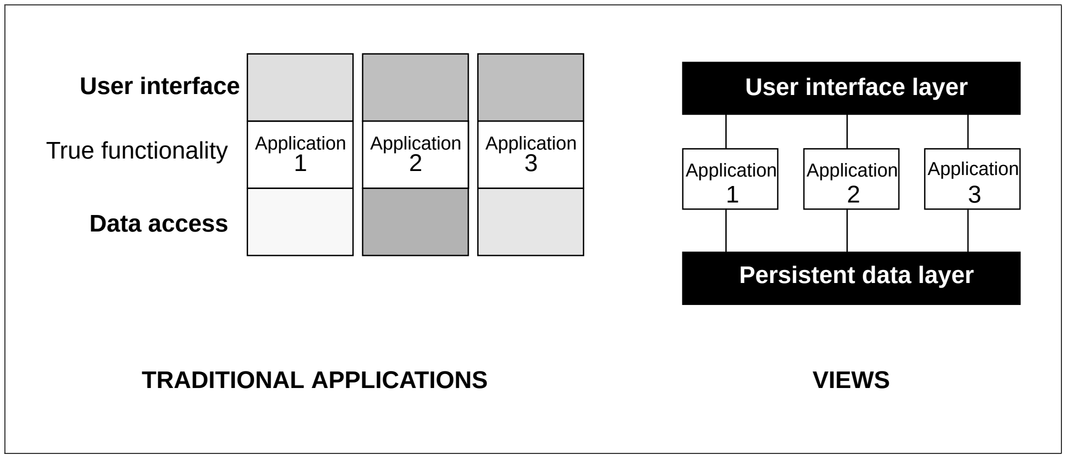


Figure 8. Rather than requiring that each application define its own data-formats as is the case with traditional applications, Views supplies a data-layer that takes care of external data-formats and where objects are stored. This allows any application to import objects from other applications without losing information about their structure and without having to know about files or even the existence of external storage.

- Sequences (lists)
- Compounds (tuples)
- Unions
- References
- Likenesses (type synonyms).

For instance, a simple clock contains three fields, the hour, the minute, and the second:

```
Type clock = compound(h: hours, m: minute, s: seconds)
```

where `hour`, `minute` and `second` are types defined elsewhere. Additionally a data-type declaration must specify the presentation:

```
Type clock = compound(h: hours, m: minute, s: seconds)
presented row[h, ":", m, ":", s]
```

What follows the word ‘presented’ is any expression yielding a ‘presentation’; in this case the function `row` takes a list of objects and takes their presentations, and joins them together horizontally. So, depending on the presentations of `hour`, `minute` and `second`, when visited, a clock might look like this:

```
12:21:30
```

It can immediately be seen that even though presentation of an object is conceptualised as a single invariant between object and presentation as in Figure 6, the system can actually break it down into lower-level invariants (see Figure 9.), giving the extra advantage of automatic screen update optimisation: if only the seconds field changes, only the seconds part of the presentation needs to be changed.

Along with `row`, there is a function `hang` that joins presentations vertically. So the presentation:

```
presented hang[h, "--", m, "--", s]
```

would give something like:

```
12
--
21
--
30
```

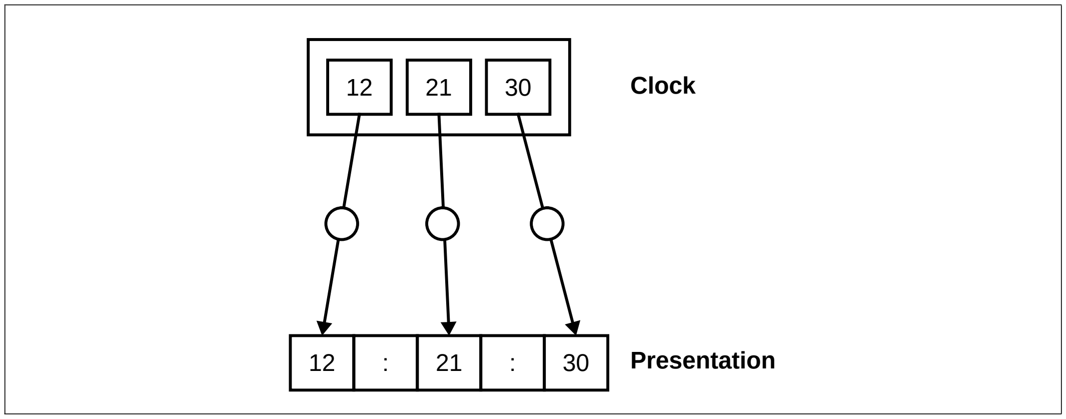


Figure 9. Splitting a high-level presentation invariant into low-level ones.

7 Invariants

Of course, the above definition of the type `clock` only defines the structure and look of the clock: it doesn't tick yet. To do this some invariants need to be defined between objects. The general structure of an invariant is:

```
    expression = expression
```

for instance:

```
    a = b + 10
```

Note that because invariants are two-way, this example is equivalent to:

```
    a - 10 = b
```

For the clock, we need the help of a system supplied object: the system seconds. This is a simple document (an integer) that represents the number of seconds since the beginning of 1970; the system updates it each second. We have to build the necessary invariants between this and our clock:

```
    import system:secs
    clock now
    now.s = secs mod 60
    now.m = (secs ÷ 60) mod 60
    now.h = (secs ÷ 3600) mod 24
```

This code imports the document `secs` from the workspace `system`, creates an instantiation of the type `clock`, and then defines the necessary invariants between the two. This could alternatively have been written using a *compound display*:

```
    import system:secs
    clock now = clock(h: (secs ÷ 3600) mod 24,
                      m: (secs ÷ 60) mod 60,
                      s: secs mod 60)
```

8 Graphics

Graphical objects in Views are treated in the same way as other types of objects. For instance, an object of type `line` contains only a number, its length. The fact that it appears on the screen as a line is purely a presentation issue (an inbuilt system property in this case). Other attributes, such as position, orientation, or colour are properties of all graphical types, and not particularly of lines.

For instance, suppose we wish to make a graphical clock. The data type is the same as the type `clock` we defined above, since the information content is the same; only the presentation changes:

```
type gclock = like clock
presented circled(combine[decor; hhand; mhand; shand])
```

Here we see the use of the function `combine`; this works like `row` and `hang`, except that it combines the presentations on top of each other as it were. The function `circled` puts a circle around any graphic.

The second hand `shand` can be defines as follows:

```
shand = line(slength) rotated (s × 6)
```

for some value of `slength`. The function `line` yields a line object as described above; the function `rotated` rotates any graphic by the given amount.

The minute hand looks the same:

```
mhand = line(mlength) rotated (m × 6)
```

The hour hand is only slightly more complicated, since its rotation should partly depend on the minutes as well:

```
hhand = line(hlength) rotated ((h mod 12) × 30 + m ÷ 2)
```

Finally `decor` is any decoration we wish to add to the clock.

To make a graphical view of the existing ticking clock, we only have to say:

```
gclock gnow
gnow = now
```

and on visiting this document, we would see something like Figure 10.

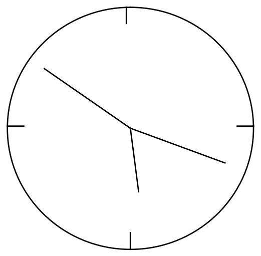


Figure 10. The graphical clock

9 Two-way invariants

It was mentioned above that invariants are in general two way. At first sight this would appear to only be possible for invertible functions, and not with functions like ‘mod’ used in the above example. However, in Views, functions are more generally invertible. Take as an example, the function of selecting a field of a compound, which is in classical mathematical terms non-invertible. How this is implemented in Views can be visualised in Figure 9: if a field is selected from an as yet unset compound, the compound is created and the other fields are marked as ‘unset’, and the invariant is set up between the field and its target. If either end of the invariant gets changed, the other end gets changed to match.

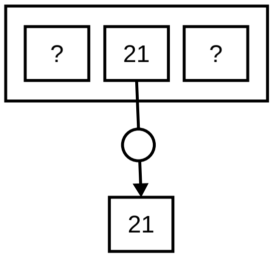


Figure 11. The invariant selecting a field of a compound.

Now we can use this fact to define ‘mod’. Mod is not invertible, but the function ‘divmod’ which returns a compound of two fields containing the (integral) divisor, and the modulo is, so the Views ‘mod’ function can be defined as:

```
function a mod b = (a divmod b).mod
```

which selects the mod field of the resulting compound.

10 Programs

As mentioned above, Views programs reside in workspaces. However, rather than being held internally as text objects, the programs are Views objects in themselves. For instance, an invariant like:

```
now.s = secs mod 60
```

is held internally as something like:

```
invariant(expr(selection(expr(id("now")), id("s"))),
          expr(operator(id("mod"), expr(id("secs")),
                                   expr(const("60")))))
```

This is in fact just the ‘abstract syntax’ of the invariant, and the presentation of this object would then add the ‘concrete syntax’. As an example, the data-type `invariant` can be defined as something like:

```
type invariant = compound(left: expr, right: expr)
presented row[left, " = ", right]
```

This means that the precise details of how Views programs look are unimportant. It also means that Views programs are ‘translated’ into an executable form (i.e. a network of invariants) using invariants themselves. In fact, programming in Views is just another Views application! At the top level there is an invariant that equates the program with the network of invariants implementing it, as in Figure 12.

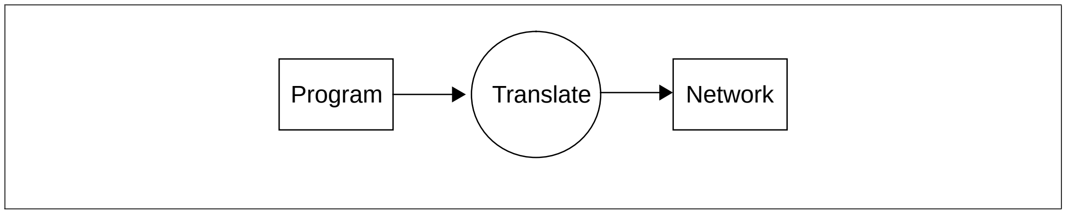


Figure 12. Translating a program.

This high-level invariant is of course split up into lower-level invariants. For instance, consider the translation of a simple identifier “now”. This identifier has to be searched for in the local ‘environment’ (a collection of named locations), to give the location referred to (creating it if necessary). This gives the picture in Figure 13.

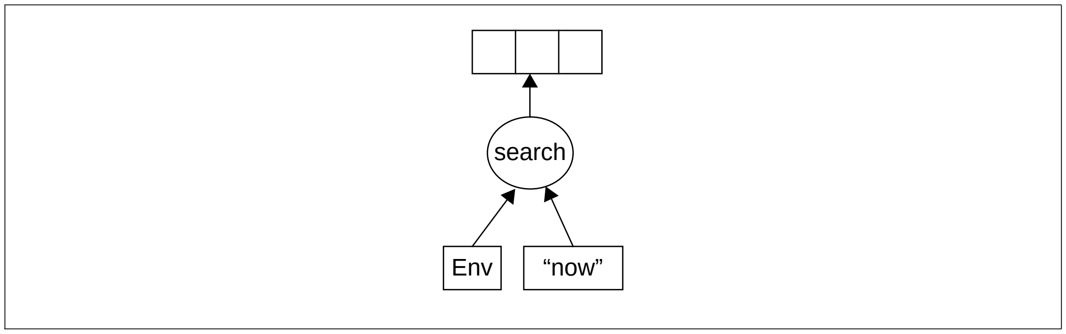


Figure 13. Translating the identifier ‘now’

A more complicated expression, like a selection such as “now.s” is pictured in Figure 14. Here there is a new invariant going from the object found, yielding its type, which is used for the search of the field “s”. This in its turn yields an offset for the true selection.

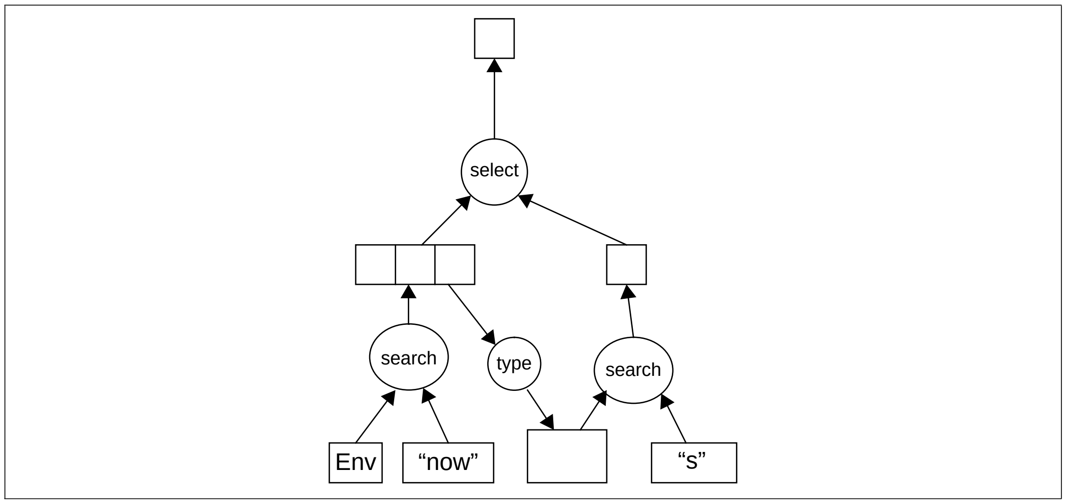


Figure 14. Translating the selection ‘now.s’

Finally an invariant such as “now.s = secs mod 60” is presented in Figure 14. You will notice that there is a strong correspondence between the abstract syntax tree for the invariant and the corresponding network of invariants, and it should be clear which parts of the network are linked (by invariants) to which part of the syntax tree.

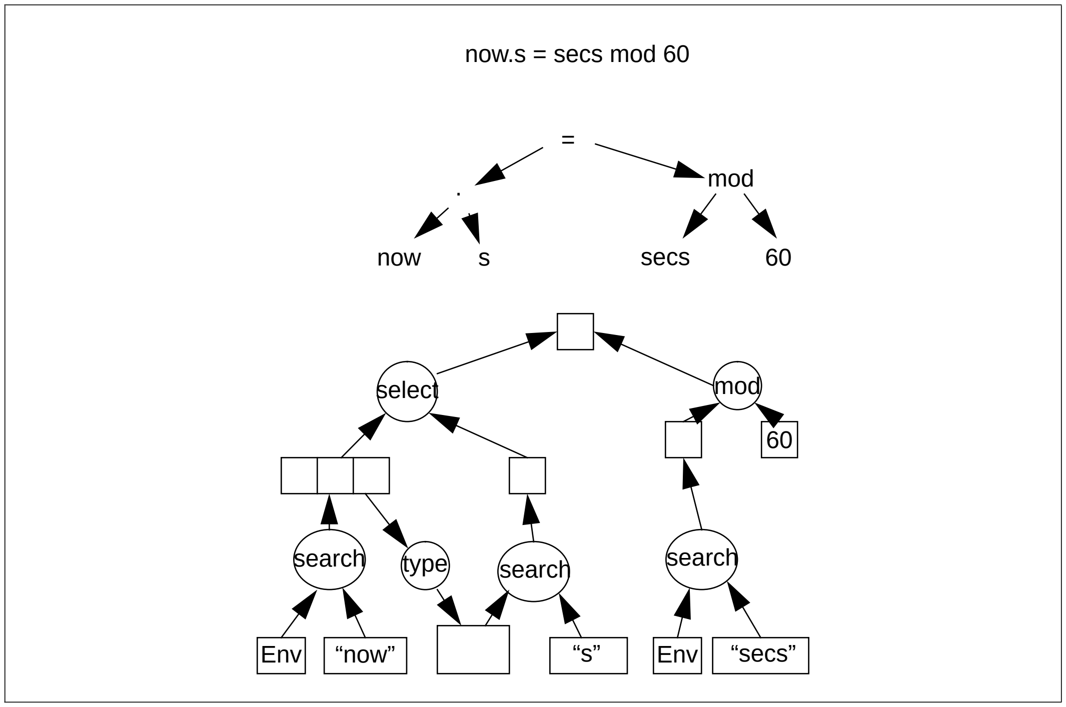


Figure 15. Translating an invariant

As mentioned, these invariants are for translating parse trees into the network of invariants implementing the code. Other work has been done (designed but not yet implemented) for translating the external representation of the parse tree (the presentation) back into a parse tree using invariants purely based on the presentation information for the types involved, in other words making the presentation invariants two-way. This is described in [8].

11 Open architecture

The fact that the whole system is defined in terms of Views objects that are accessible and changeable means that many aspects of the system which are traditionally hardwired are tailorable to the user in Views. We have already mentioned the representation of programs, but such things as menus and shortcuts are also tailorable to the user in Views, even on the fly.

The reader will have noted that Views is actually an incrementally evaluated functional system, and that everything visible is just the result of applying functions to objects. In fact this is true right down to the lowest level. The system has no true concept of the 'events' that are present in traditional windowing systems. Within the Views kernel events are just added to a buffer from which leads an invariant that causes them to be distributed to the interested parties, by appending them to buffers which in their turn are attached to invariants. In this way the whole system is just the result of evaluating the effect of all events that have come in on the initial state of the system. While not everything in the actual implementation is purely functional (for instance the events are thrown away after being handled), the non-functional parts can be modelled functionally and the difference can be seen as optimisation. Further treatment of the low-level use of the Views model can be found in [9].

12 Conclusions

Programming for applications in ‘traditional’ windowing environments is difficult and time consuming. While tools such as graphical and user interface toolboxes, and User Interface Management Systems can help the programmer, applications remain unwieldy, and isolated from each other, adding to the problems for the user of the applications. It is our belief that the problem lies in the degree of abstraction offered by existing windowing systems; imagine the programming and user problems that would exist if current operating systems didn’t offer hierarchical file stores, but only ways to access the bytes on disks. Views addresses many of the problems experienced by programmers and users of existing windowing applications by unifying the approach to applications and using a very simple yet powerful invariant mechanism widely across the system, yielding very great gains in programming time and ease of use.

13 References

[1] Brad A. Myers and Mary Beth Rosson, *Survey on User Interface Programming*, in *Proceedings of CHI, 1992* (Monterey, California, May 1992), ACM, New York, 1992, 195-202.

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

[3] R. M. Stallman*, EMACS: the extensible, customizable, self-documenting display editor*. In D. Barstow *et al*., *Interactive Programming Environments*, McGraw-Hill, New York, 1984.

[4] G.L. Steele and G.J. Sussman, *Constraints,* APL Proceedings, APL Quote Quad, June 1979, 208-225.

[5] Wm Leler, *Constraint Programming Languages, Their Specification and Generation*, Addison Wesley, 1988.

[6] J. van de Graaf, *Towards a Specification of the B Programming Environment*, CWI Report CS-R8408, CWI, Amsterdam 1984.

[7] Lambert Meertens, Steven Pemberton, Guido van Rossum, *The ABC Structure Editor*, CWI Report CS-R9256, CWI, Amsterdam,1992.

[8] Job Ganzevoort, *Maintaining Presentation Invariants in the Views System,* CWI Report CS-R9262, CWI, Amsterdam, December 1992.

[9] Eddy Boeve, *Modelling Interaction Tools in the Views Architecture,* CWI Report CS-R9261, CWI, Amsterdam, December 1992.