A Hyperlinked Persistent Software Development Environment

Alan Dearle

Department of Computer Science,
University of Adelaide,
G.P.O. Box 498, Adelaide,
South Australia 5001,
Australia.

al@cs.adelaide.edu.au

Chris Marlin

Discipline of Computer Science,
Flinders University of South Australia,
G.P.O. Box 2100,
Adelaide,
South Australia 5001,
Australia.

marlin@cs.flinders.edu.au

Philip Dart

Information Technology Division,
Defence Science and Technology Organisation,
P.O. Box 1500,
Salisbury,
South Australia 5108,
Australia.

philip@itd.dsto.oz.au

Introduction

Traditionally, software development environments have been constructed on top of conventional operating systems using the file store as the only persistent storage facility. Such an approach forces the programmer to treat these persistent data structures in a fundamentally different way from other data structures. In a system with *orthogonal persistence* [1], any data structure may persist and thus outlive the execution of a program. Under such circumstances, it is no longer necessary for the programmer to treat long-term and short-term data structures in different ways. Thus, the introduction of orthogonal persistence removes many of the discontinuities inherent in most software systems.

In this paper, we will present our initial ideas on the creation of an integrated environment for the development of persistent systems. Naturally, this development system will, by necessity, be resident within the total persistent environment. As such, a persistent object store provides the basis for the integrated programming environment. The result is an environment which differs significantly from those which have been developed as an adjunct to traditional programming practices in a file-based operating system. In this paper, we present our initial ideas on such an environment. In particular, we focus on potential uses for hyperlinking in the environment.

A persistent object store

The persistent object store is comprised of a graph of interconnected objects, reachable from one or more persistent roots. The data in the store may be simple or complex, including executable programs, source programs, specifications, programmers' notes, pictures (diagrams), simple values, graphs of sample data and interdependency information. Clearly, the persistent store subsumes the role of both a file system and an object library. However, in order to utilise the persistent object store, it is necessary to impose structure upon it. A store containing many useful objects which cannot be found would clearly be next to useless.

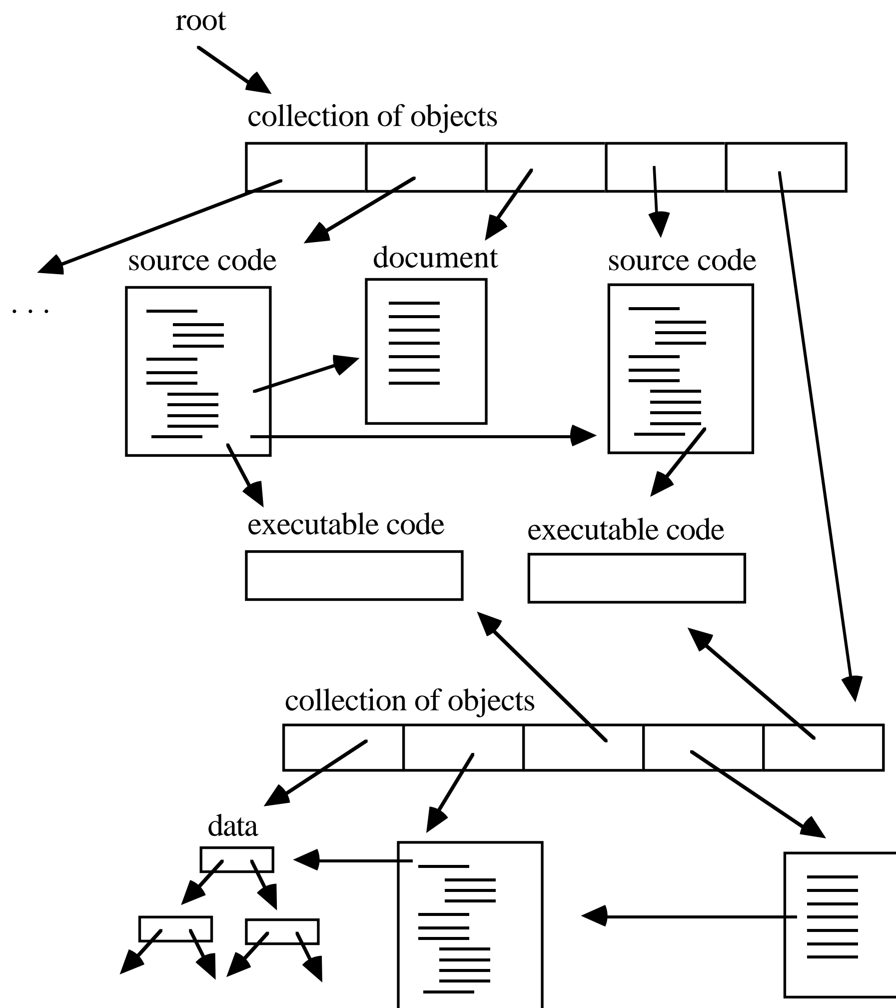

Figure 1. A persistent store containing software and other objects.

In order to overcome problems of locating useful data, it is necessary to thread the persistent store with a web of interconnections, as shown in Figure 1. This interconnection graph contains comments and descriptions of data held at the various nodes in the graph. The web's main function is to provide regularity over the potentially chaotic mass of data and corresponds to a hyperlinked database overlaying the persistent store (c.f. [2]), interconnecting certain key objects in it. With a regular structure, it is possible to provide high level mechanisms for searching the persistent store to find items of interest to the user. This task may be viewed as being similar to performing queries over a traditional database system. It is important to note that efficient database queries are only possible over a regular structure, and that efficient queries are necessary if the system is supporting a large development environment.

Navigating the object store

Performing queries to search for particular items of data may be viewed as a special case of something we call *browser technology* [3]. In this context, a browser is a program which traverses a graph performing some action as it goes. We have already demonstrated the usefulness of browsers for locating and displaying graphs of various kinds of data. Since the persistent store may potentially contain an infinite number of different types of objects, a browser must be capable of locating and displaying objects whose type had not been known when the browser was written.

We propose to apply the browser techniques already developed to the software reuse problem. In particular, we assert that object browsers may be employed to browse the hyperlinked database mentioned earlier. In such a system, users will find reusable software by following the various interconnections and interdependencies contained in the object store.

Viewing and altering objects in the persistent store

As illustrated in Figure 1 above, the kinds of object discovered by the browser in our Persistent Integrated Programming Environment (PIPE) will include procedures, abstract code, simple values and graphs of sample data. These different types of objects need to be displayed in different ways. Indeed many objects benefit from being displayed in more than one way, employing the multiple views approach used by Pecan [7] and MultiView [6]. For example, trees or graphs representing the abstract syntax of a procedure might be displayed textually and/or in some graphical form.

Program components should be edited in a language-specific manner, but not using the straight-jacket of a fully-template-based editor. The underlying representation of the program components should be structured, probably tree-structured, at some level of abstraction. Thus, selecting some part of a program component amounts to selecting some subtree. For example, Figure 2 shows a Napier88 [5] program component; part of the component has been selected by the user. This part corresponds to a complete subtree in the underlying structured representation of the program component.

Because a complete subtree must be selected, it is not possible to select only the keyword "**type**"; selecting this keyword would indicate that the entire construct highlighted in Figure 2 is to be selected. However, it is possible to select the keyword "**string**", since this corresponds to a complete subtree (although a simple one).

```
type domicile is structure( name, location : string )

type animal is structure( name : string ;
                          home : domicile )

let smallPond = domicile( "pond", "Adelaide" )

let afrog = animal( "tadpole", smallPond )

afrog( name ) := "frog"

use PS() with bigPond : domicile in
                  afrog( home ) := bigPond

in PS() let Kermit = afrog
```

Figure 2. A Napier 88 program component with part of it selected.

Associated with program components are various kinds of documentation. Some of this documentation is typically included in the program code as comments, whereas the remainder may be found in specification documents, user manuals, internal documentation, and so on. In an ideal integrated programming environment, there is no need to distinguish between these various kinds of documentation: all can be supported within the environment. If desired, some of this documentation can be selectively turned into a printed document (e.g., a specification document describing the system).

Ideally, all documents should be manipulated using a document editor whose functionality corresponds to a modern word-processor (multiple fonts, flexible paragraph layout, good drawing facilities, and so on).

Hyperlinking

The key to integration between program components and their associated documentation is to permit various kinds of hyperlinks between them. The most obvious example is to permit a link between a subtree in the structured representation of a program component and some part of the documentation.

Figure 3 shows examples of instances of possible document and program editors as displayed using the object browser. In the upper left of this figure is an instance of a document editor which is being used to view/manipulate a specification document called **Force.spec**. Below it is another document editor instance; the latter contains a document entitled **Force.doc**, which represents documentation on a program component. Finally, on the righthand side of the figure is an instance of the program editor; this instance contains a program component called **Force.npr**.

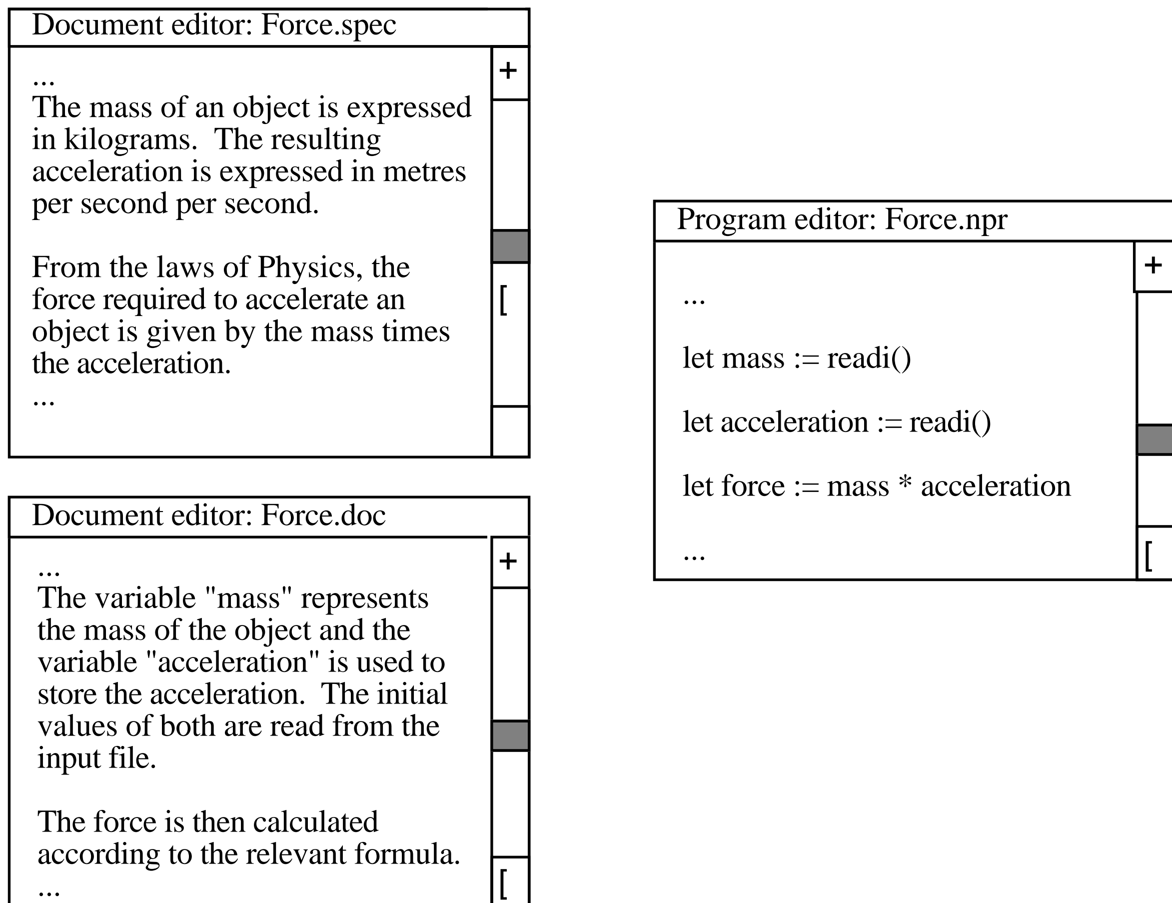

Figure 3. Document and program editors.

Various kinds of links might usefully be established between parts of the contents of the three editor instances in Figure 3. For example, links between the specification in **Force.spec** and the code in **Force.npr** serve to indicate the relationship between the original specification and the code to implement aspects of that specification, and vice versa. Thus, having selected the definition of "acceleration", as shown in Figure 4, it is possible to establish a link to the relevant part of the specification (i.e., the second sentence); this link could then be used later to move from the definition of the variable to its explanation in the specification. Conversely, if the second sentence in the specification or some part of it were to be selected, then it should be possible to link this sentence to the definition of "acceleration". Notice that the parts of the program source text to/from which the hyperlinks are established correspond to complete subtrees in the underlying representation.

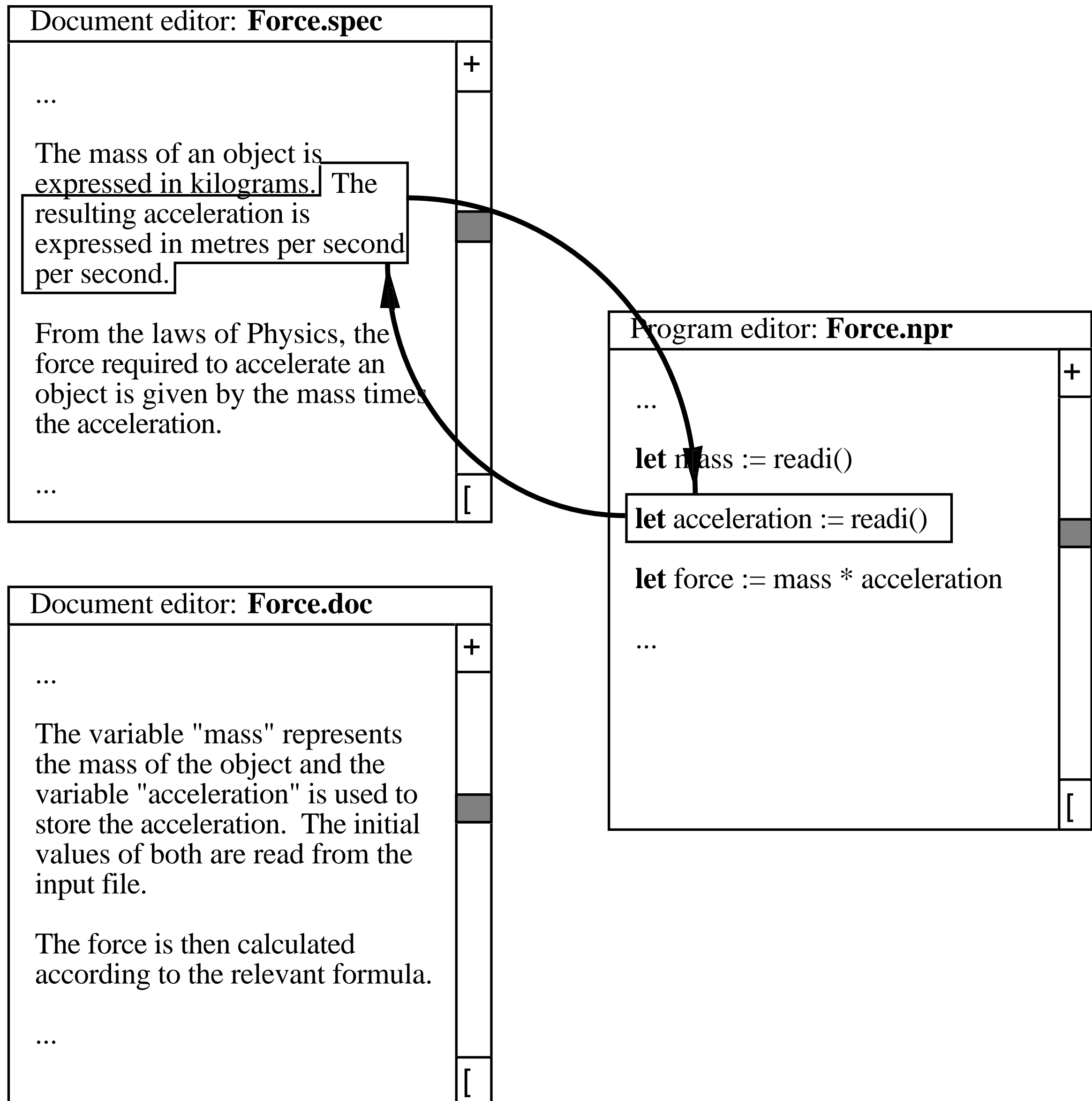

Figure 4. Links between specification and source code.

Another useful kind of hyperlink is that between program source code and its internal documentation – such a link subsumes the role of comments in a traditional programming language environment. As an example, consider Figure 5, where links exist between a part of the program component in the program editor and a part of the internal documentation file. So far, the links have all been one-to-one, but this may not be so, as illustrated in Figure 6.

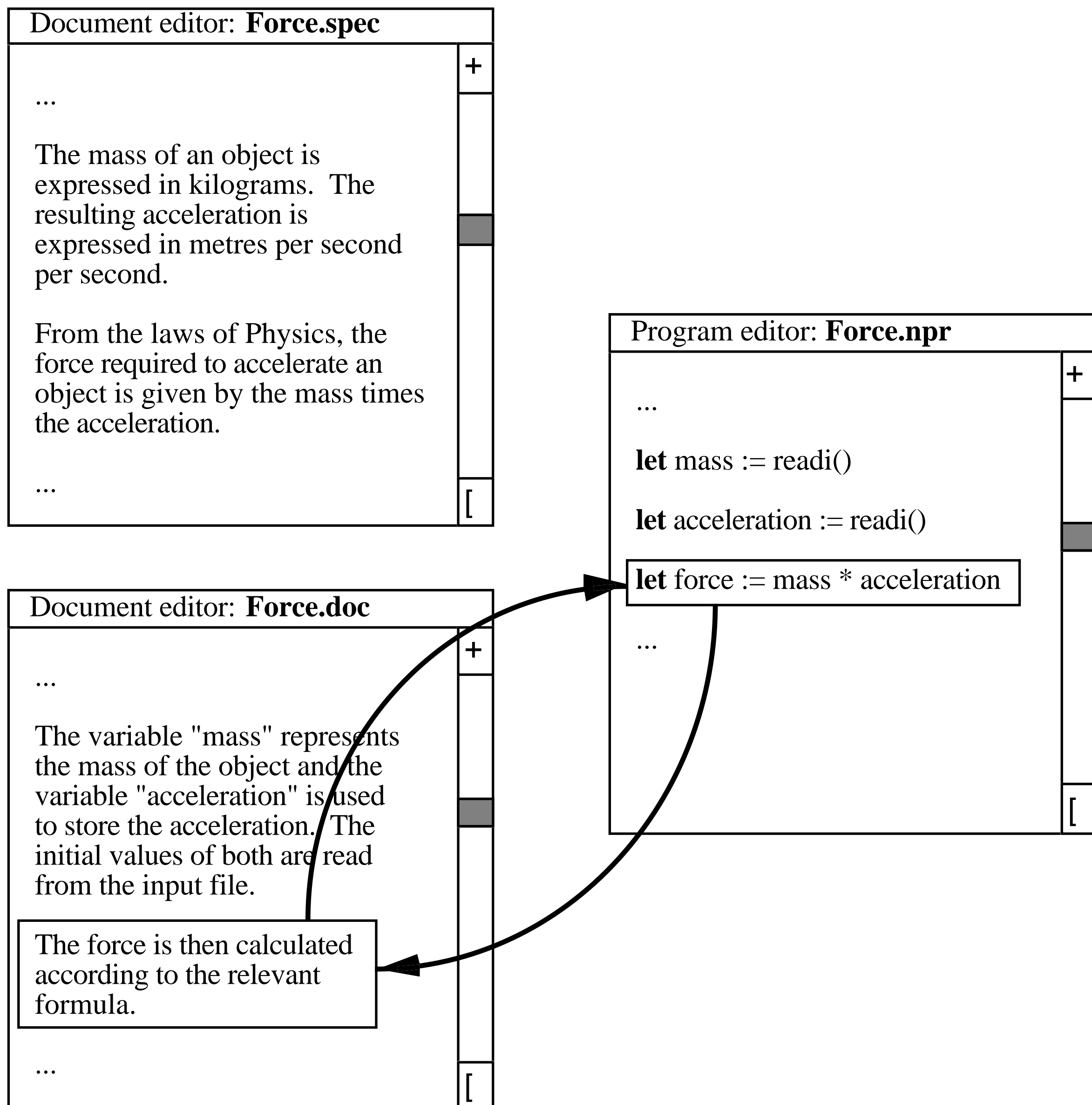

Figure 5. Links between source code and internal documentation.

In Figure 6, the documentation relating to the fact that the initial values of the variables "mass" and "acceleration" are read from the input file is to be found in the second of the sentences shown in the lower instance of the document editor. Thus, selecting either of

the initialisation parts of the first two clauses shown in the program editor instance allows hyperlinks to the relevant documentation to be followed.

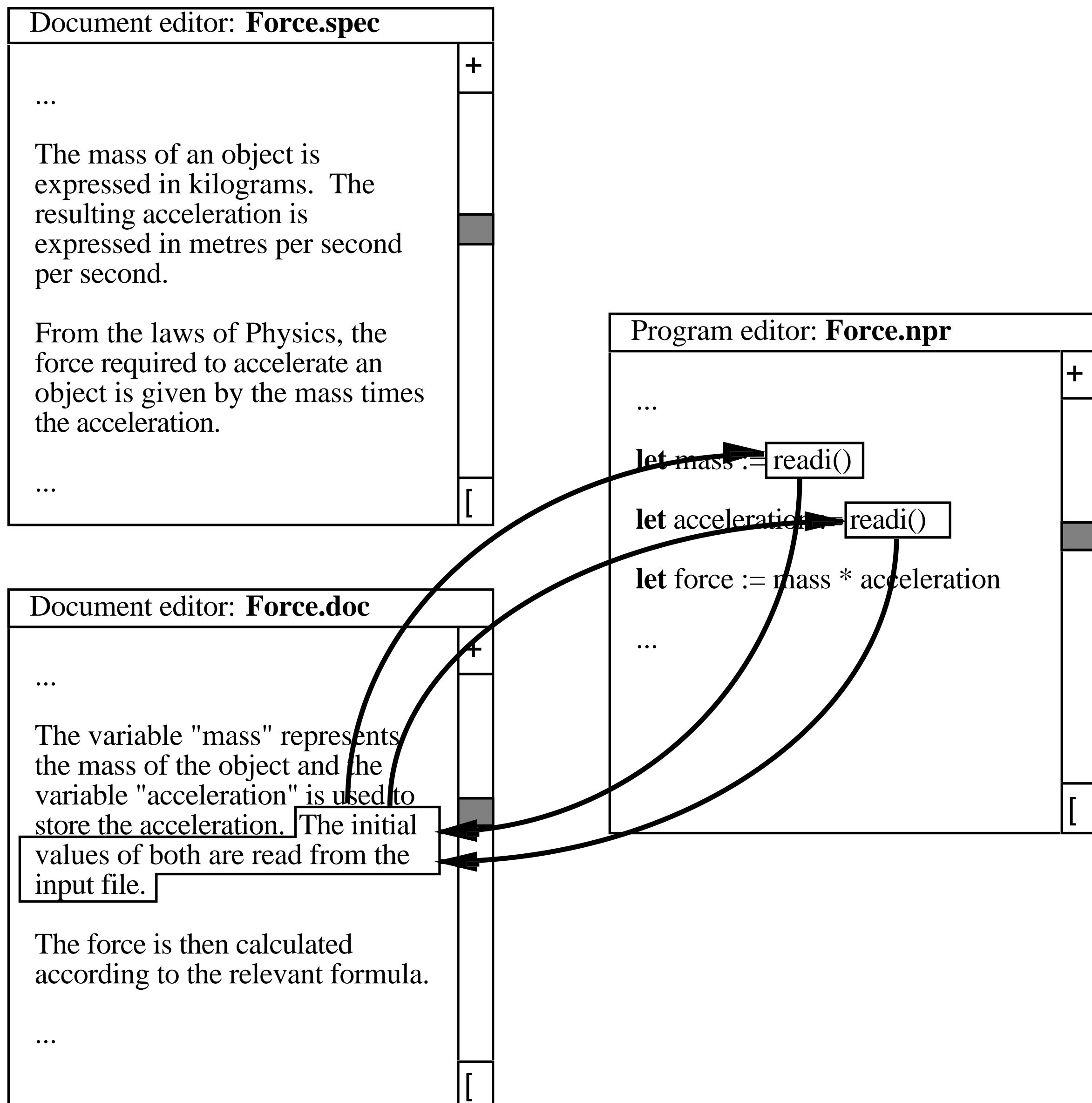

Figure 6. Many-to-one links between source code and internal documentation.

The reverse links (from the sentence in the documentation into the source code) are more problematic from the point of view of the user interface, since there is no single place to go as a result of selecting the sentence. Some kind of "iterator" mechanism may be called for; this would allow the user to cycle through the relevant parts of the program component's source code. Thus, in the case of Figure 6, the user interface might take the user to the first "readi()", but there should be some simple means to see that there are other possibilities and to move on to the next such possibility (and then back to the first

again). Depending on the context, this may be facilitated by merely highlighting all of the possibilities or by allowing the user to move to the next in the list by a simple keystroke.

Finally, there is value in having hyperlinks between documents. For example, it is possible to imagine hyperlinks between a specification document and some internal documentation. This is illustrated in Figure 7, where a phrase in the internal documentation is linked to a paragraph in the specification document. Other links of this kind may be useful, such as between the phrase "the mass of the object" in the internal documentation and the first sentence shown in the upper document editor instance.

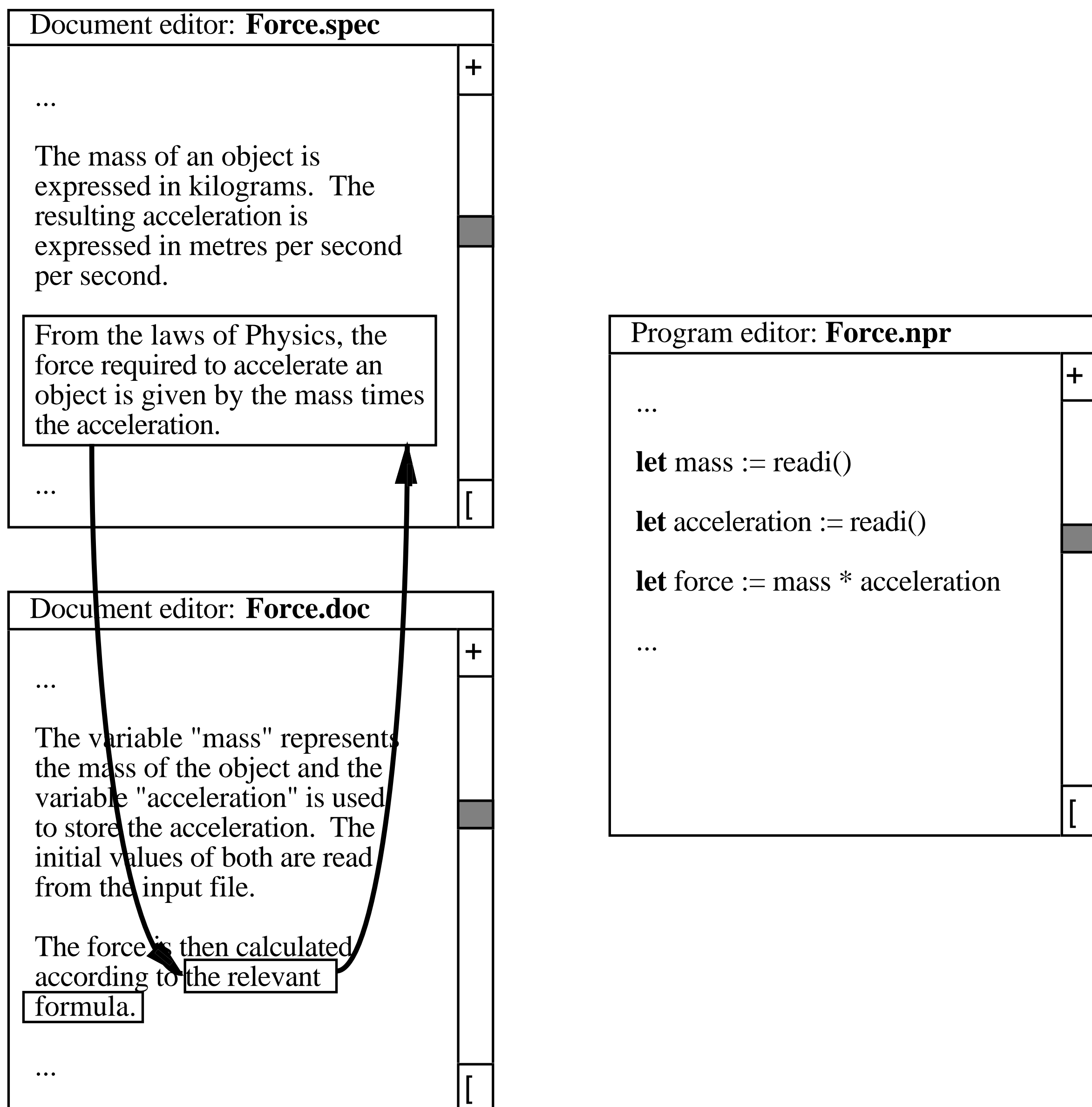

Figure 7. Links between specification and internal documentation.

It is also be possible to hyperlink various program components. To illustrate the possibilities in this area, we will be using the example in Figure 8. This shows two program components (Napier88 procedures called "p1" and "p2", in this case) and their associated internal documentation.

As illustrated in Figure 8 below, various hyperlinks are possible between a program component and its corresponding documentation. For example, one would expect a link between the component "p1" and the whole of its documentation; thus, making the selection shown in Figure 9 would enable the user to move to the corresponding documentation, i.e. to the upper left instance in Figure 9. If there were no instance of the document editor containing the corresponding documentation, one would be created as a side-effect of this operation.

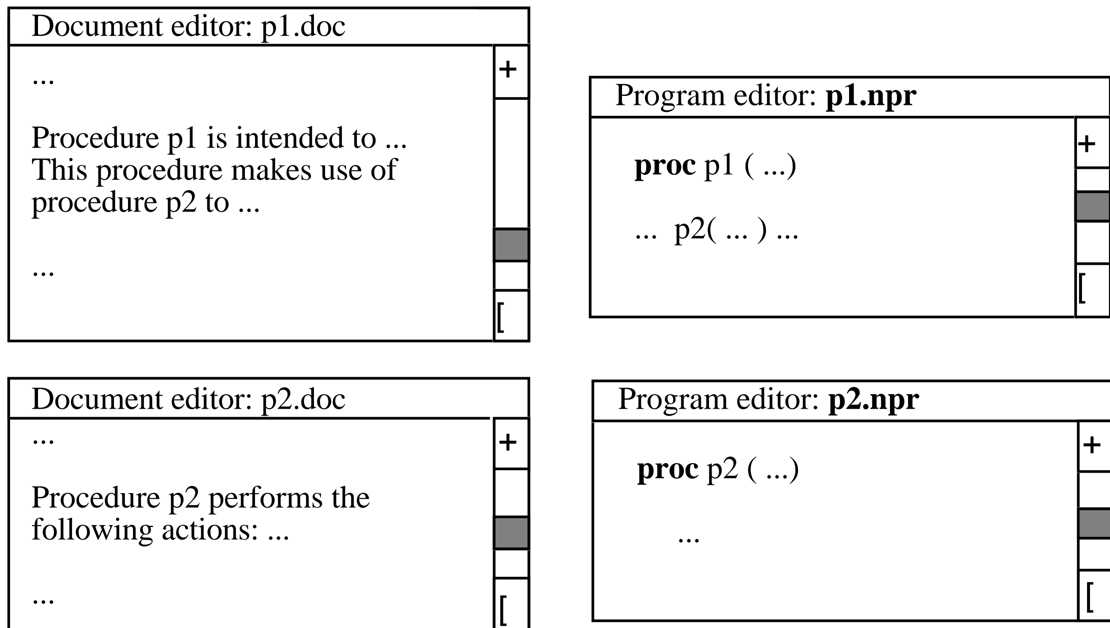

Figure 8. Two program components and their associated internal documentation.

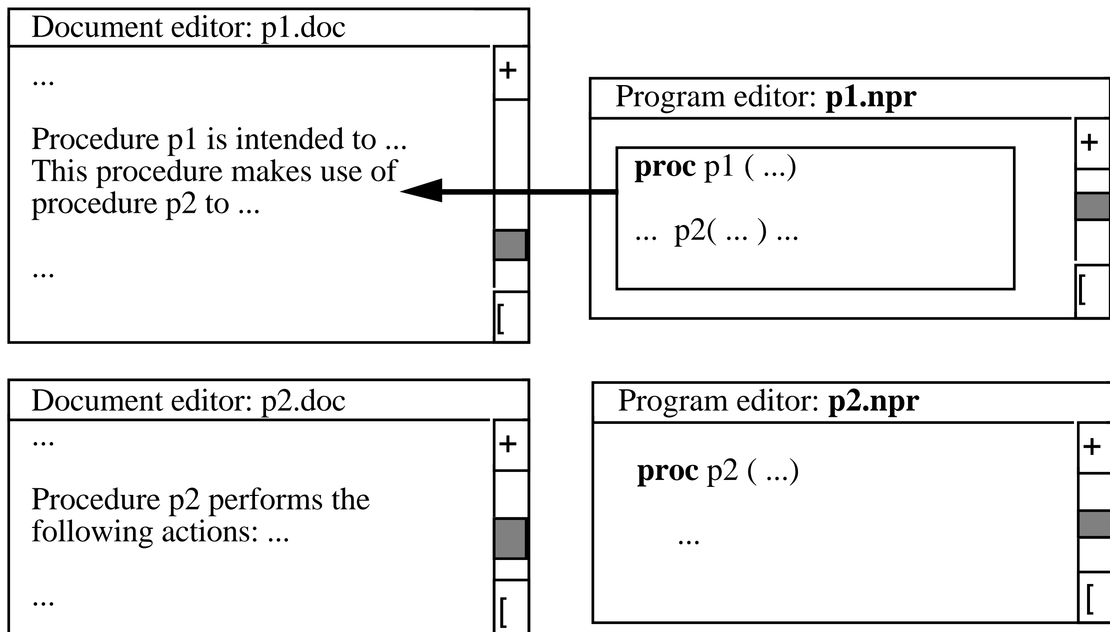

Figure 9. A link from a program component to its documentation.

In addition, it should be possible to follow links to the documentation for other program components. For example, the reference to "p2" in the upper right editor in Figure 8 could be used to move to the corresponding part of the the documentation for "p1" or to the documentation for "p2" itself; both of these possibilities are shown in Figure 10.

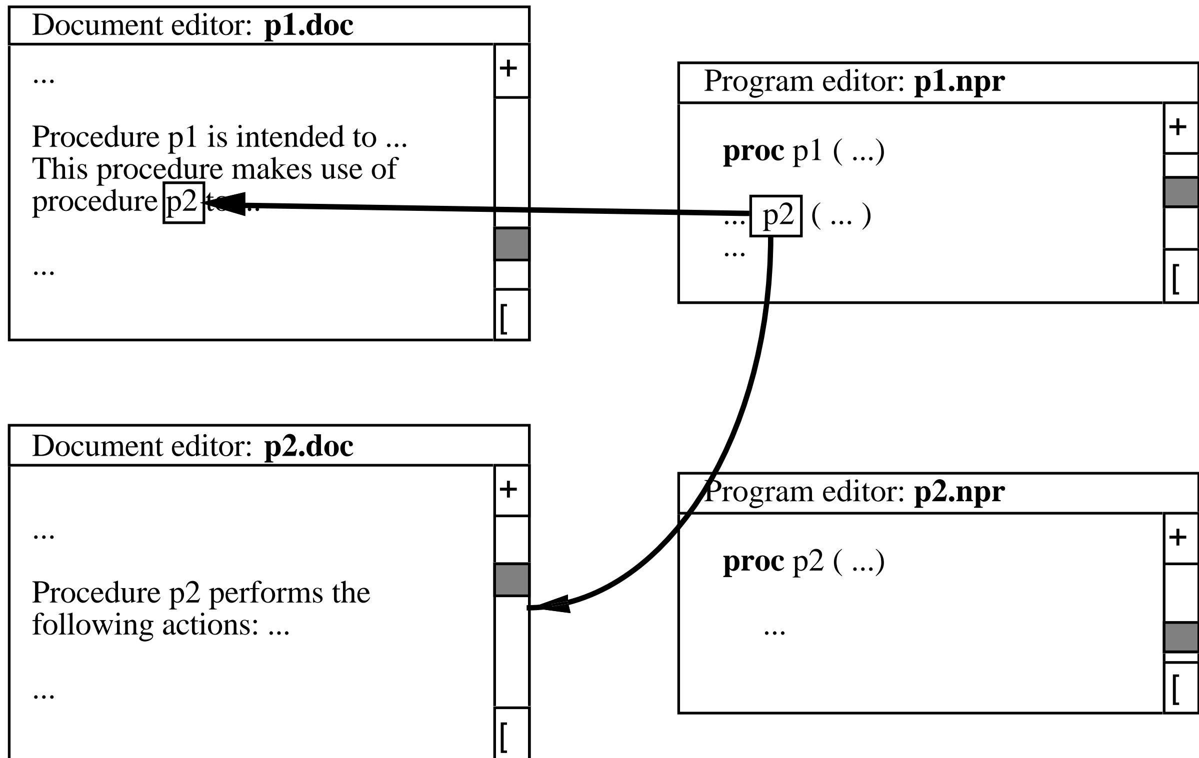

Figure 10. A link from a program component to documentation for more than one program component.

It should also be possible to establish links between documents relating to different program components. For example, the documentation for "p1" above refers to "p2"; it would be useful to be able to use the reference to "p2" in the documentation for "p1" to be able to move to the documentation for "p2", perhaps invoking an appropriate instance of the document editor if one does not already exist. This is illustrated in Figure 11. Conversely, it should be possible to start with the documentation for "p2" and cycle through the documentation for all the program components which make use of "p2". Ideally, such links should be established automatically from an analysis of the code concerned and a knowledge of the binding between program components and their documentation.

Related to the above discussion of possible links between the documentation corresponding to different program components is the question of links between program components. Two kinds of such links can readily be identified: "uses" and "used-by" links. Both of these should be maintained automatically, rather than inserted by the user. These two kinds of link are illustrated in Figure 12, which shows a "uses" link from a reference to "p2" within "p1" to the definition of "p2", and a "used-by" link from the definition of "p2" to where it is used in "p1". Once again, it should be possible to cycle through all the places that "p2" is used. As before, following a link may cause the creation of a new instance of a program editor.

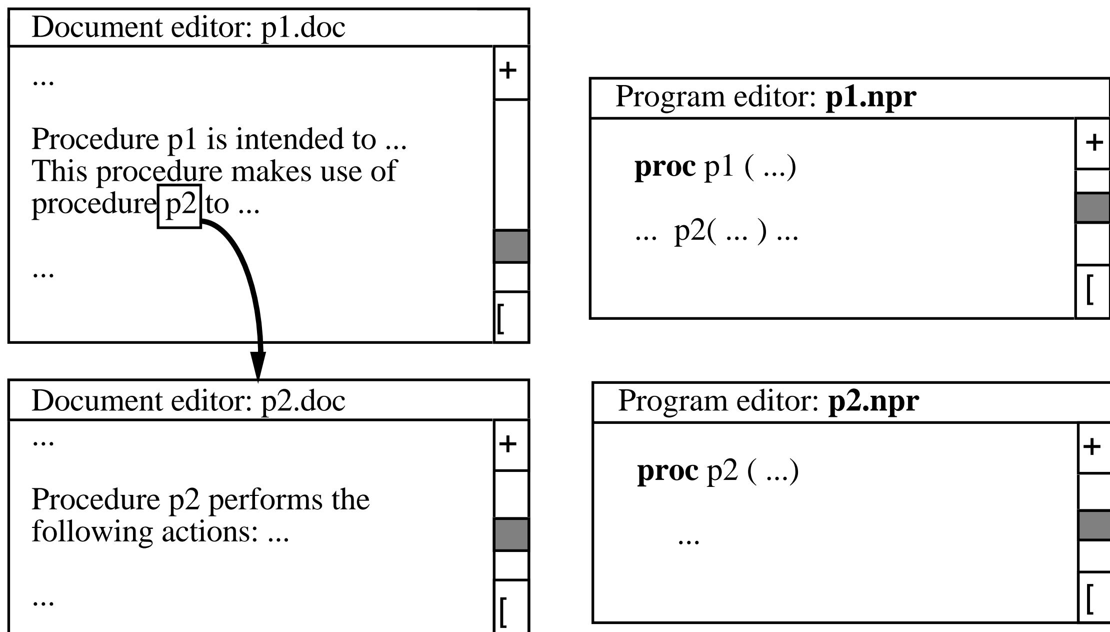

Figure 11. A link between the internal documentation for different program components.

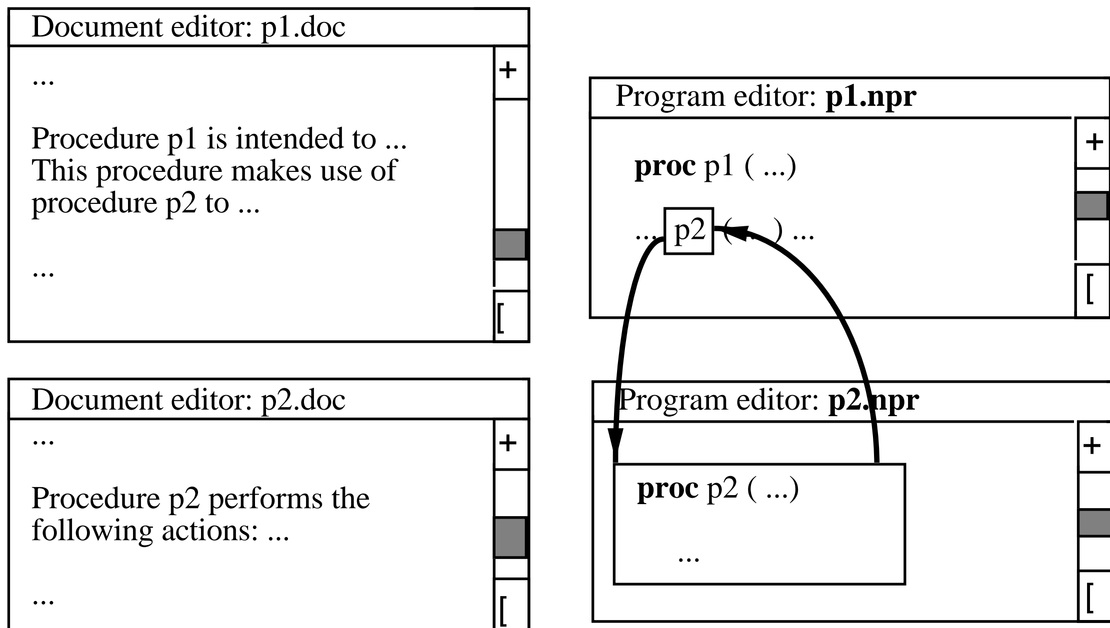

Figure 12. Links between program components.

Conclusions and future work

The manner in which hyperlinking may be used in a persistent integrated programming environment has been described. These uses include linking specification and documentation to the corresponding program source code, and various kinds of links between related parts of the program source code. There are no doubt many other ways in which hyperlinking could be used in such a programming environment.

The construction of a persistent integrated programming environment based on the above ideas is in its early stages. Initially, the thrust has been to concentrate on document and program editors, and the tools capable of generating them. Independently of this effort, work has been continuing on browser technology and a programmable Napier object browser has been completed [4]. Some ideas for hyperlinking between objects in the persistent object store have also been prototyped. Once a suitable framework of such facilities has been constructed, this framework will be used in the investigation of research issues related to the kind of support which is helpful during the construction of persistent software systems.

References

1. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp. 360-365, 1983.

2. Bigelow, J. "Hypertext and CASE", *IEEE Software*, Vol 5, No 2, March, 1988.

3. Dearle, A. and Brown, A. L. "Safe Browsing in a Strongly Typed Persistent Environment", *The Computer Journal*, vol 31, 6, pp. 540-545, 1988.

4. Farkas, A. "Aberdeen: A Browser allowing Interactive Declarations and Expressions in Napier88", University of Adelaide, 1991.

5. Morrison, R., Brown, F., Connor, R. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews, PPRR-77-89, 1989.

6. R.A. Altmann, A. N. H. a. C. D. M. "An Integrated Programming Environment Based on Multiple Concurrent Views", *The Australian Computer Journal*, vol 20, 2, 1988.

7. Reiss, S. P. "Graphical program development with PECAN program development", *SIGPLAN Notices*, vol 19, 5, pp. 30-41, 1984.