Computational Reflection and CSCW Design

Paul Dourish

Technical Report EPC-1992-102

Rank Xerox Research Centre
Cambridge Laboratory
61 Regent Street
Cambridge CB2 1AB

Tel:+44 1223 341500
Fax:+44 1223 341510

Computational Reflection and CSCW Design

Paul Dourish
Rank Xerox EuroPARC
Cambridge, UK

dourish@europarc.xerox.com

March 13, 1992

"Most disquieting reflection of all, was it not bad form to think about good form?"
— J.M. Barrie, "Peter Pan"

Abstract

Designing computer systems to support collaborative work is hard. One of the major reasons for this is the requirement that such systems be *flexible*, able to adapt themselves to different users or groups, able to track changes in group behaviour over time, and able to function efficiently in different computational and infrastructural environments. For the designer of a toolkit for building CSCW applications, these problems are heightened, since a toolkit needs to be applicable to a range of applications, each of which require flexibility along these different dimensions. This paper proposes a toolkit based on the theories and techniques of *computational reflection*, which enable the implementation of malleable, open-ended systems with dynamic properties which are particularly applicable to collaborative systems design.

1 Introduction

This paper concerns the applicability of the theories and techniques of computational reflection to design in the domain of computer systems to support cooperative work. Of necessity, this paper contains a large amount of background information on computational reflection and, to a lesser extent, CSCW design; both of these are crucial if we're to make the link.

I will begin by describing the general model of computational reflection, and, through a series of examples, make this description more concrete and give a flavour of how reflection can used to build open-ended and flexible systems. We will then discuss flexibility as the major design problem for CSCW, and describe how reflection might help make flexible CSCW systems considerably easier to design.

2 What is Computational Reflection?

In the psychological domain, "reflection" refers to a process of introspection and analysis. By analogy, *computational reflection* is the ability of a computational system to "reason" about its own behaviour (and then to change it). The key elements are, first, a *representation* which the system holds of its own behaviour; and second, a *causal connection* between that representation and the actual behaviour. The causal connection is two-way; not only are changes in the behaviour reflected in equivalent changes in the representation, but changes in the representation will also cause changes in the behaviour.

To help make this general definition more concrete, it's worth considering why reflection might be useful in computational systems. The need for reflection was perhaps first recognised in the field of artificial intelligence; it seems reasonable to presume that an autonomous agent acting in a world will be better equipped if it has the ability to reflect on its own behaviour and make appropriate modifications in an open-ended manner. When reflection moved into the realm of programming language design, a whole new area opened up. By incorporating a reflective model of computation into a programming language, tools such as debuggers and trace packages, which have always been regarded as inherently non-portable because of the need to delve into the details of language implementation, could be made portable by enabling them to act on the defined representation which is incorporated into the programming language itself. In general, then, computational reflection lends itself to the design of flexible and open-ended systems.

3 Reflection in Action: Some Examples

To show the form that this flexibility might take, we will now look at three examples. These examples are presented not only in historical order, but also from most general and theoretic, to most specific and with the clearest applications to a domain such as CSCW.

3.1 3-Lisp, A Reflective Dialect of Lisp

The first example we will consider is 3-Lisp [Smith, 1982], a reflective dialect of the programming language Lisp. This work effectively marks the beginning of the intellectual history of computational reflection. The design of 3-Lisp features not only the inclusion of reflection, but also a semantic reconstruction of Lisp, and especially its model of types and evaluation. The details of the semantic analysis and reconstruction do not concern us here, although, along with the notion of reflection, they play an important part in placing the semantics of programming languages on more stable ground. For the moment, we will consider merely the notion of reflection as presented in 3-Lisp.

The inclusion of a reflective model in Lisp can be considered as an extension of the notion of *metacircular interpretation*, which is the traditional model of Lisp implementation[1]. A metacircular interpreter is a programming language interpreter which interprets "itself". In practice, this would mean that, for instance, we might have a program, written in Lisp, which computes about some domain, such as statistical analysis. Since our computer does not understand Lisp directly, the Lisp program must be interpreted by some other program. This program may also be written in Lisp. Theoretically, this circularity can extend infinitely; in practice, we will very quickly reach the point where our Lisp interpreter has been compiled into machine instructions which can execute directly on the computer, and hence require no further interpretation.

Ignoring the issue of the level at which the metacircularity bottoms out, we can observe that Lisp, like most other programming languages, puts barriers between the implemented and implementing programs. Each of the programs involved computes entirely within its own domain (in our example, one program is concerned purely with statistical analysis, and another purely with Lisp interpretation). However, more than most programming languages, Lisp has always made certain concessions and allowed the distinction between program and language processor to be blurred slightly. From the earliest days of Lisp 1 [McCarthy et al, 1960], the function `eval` has provided a way for data structures to be interpreted as programs under programmatic control; and Scheme [Clinger and Rees, 1986] has always been marked by the presence of explicit continuations, which again provide a mechanism for capturing aspects of the execution process. So the choice of Lisp as a language to incorporate reflection is not a surprising one; it is part of a tradition.

The model of reflection in 3-Lisp is as follows:

1. Consider a dialect of Lisp which is reflective in the sense that it can appeal to a model of its own execution and can cause changes to that model which will in turn change the process of execution. This reflective ability takes the form of a mechanism by which a program can cause a Lisp expression to be executed directly by its interpreting program, and *in the context* of that interpreting program.
2. We have specified that there is an interpreting program. This interpreting program is also written in Lisp. In fact, it is written in the same, reflective dialect of Lisp, 3-Lisp. This implies that it, too, has the ability to pass Lisp expressions to *its* interpreting program.
3. Since, at each stage, we have required the ability to refer to an interpreting program, we have in fact specified that there are an infinite number of interpreting programs. Say that the original program, computing about some domain such as statistical analysis, is running at "level zero". For each program running at level *n*, let us say that it is being interpreted by another program running at level *n+1*.

This model, then, defines what Smith calls an "infinite tower of reflective processors", where each processor is interpreting the program at the level below. Fortunately for Smith, and for the rest of us, the infinite tower has a finite implementation, through the notion of a *level-shifting interpreter* [des Rivières and Smith, 1984]. While an understanding of the workings of the level-shifting interpreter are not key to the issues being discussed in this paper, they make an interesting diversion; and it *is* crucial to understand that a solution to the apparently infinite regress exists.

Informally, the level-shifting interpreter solution works as follows:

1. Consider a program in a 3-Lisp. This program will either "reflect" (*i.e.* use the reflective facilities by passing a form to the next level), or not. Call non-reflective programs "boring", and reflective ones "interesting".
2. Boring programs will execute normally—we can consider them as being standard Lisp programs. In fact, we do not need the infinite tower to execute such programs; we merely need a single interpreter.
3. Interesting programs fall into two categories. In the first category are those programs which cause infinite reflection, *i.e.* they cause an expression to be sent to the top of the tower, through the intermediate processors. Such programs are, by their nature, non-terminating; and hence there is no way to run them to completion, whatever our implementation does.

1. Lisp may be interpreted or compiled. We will refer principally to interpretation rather than compilation, although the model is equally applicable to both forms of processing.

4. We are left with programs which do reflect, but to a finite extent. Only a finite part of the infinite stack of interpreters will be involved in the execution.

5. Levels of interpretation can be created by need. That is, while a program is boring, we will merely maintain a single interpreter to run it. When that program reflects, we will *construct* the meta-interpreter (or rather, its data-structures) from values maintained in the interpreter, and then "shift" to that level to execute the reflected form. This process can proceed as long as we reflect and require new levels. When we return from the reflection, the constructed level can be torn down, until it is needed again.

Thus the level-shifting interpreter never creates more levels of interpretation than are necessary to correctly execute the program, and an infinite regress is avoided. The execution model may be infinite, but the implementation can be finite and computationally tractable.

3.2 CLOS, A Reflective Object System

The next example we shall consider is the design and implementation of CLOS[2] [X3J13, 1988], the Common Lisp Object System. There are two important points in this example: first, the integration of computational reflection with object-oriented programming; and second, the nature and role of the "meta-object protocol" [Kiczales et al, 1991].

CLOS is the object system for Common Lisp, and is now part of the Common Lisp specification undergoing ANSI standardisation. Reflection in CLOS stems from the fact that CLOS is itself a program written in CLOS; indeed, this is part of its definition. Unlike, for instance, a Lisp interpreter written in Lisp, the CLOS implementation is not hidden from the CLOS "user" (a programmer writing CLOS programs), but rather aspects of that implementation are made explicit. The programmer can then use those aspects of the implementation to extend the CLOS system. This will become more clear if we look at some examples illustrating how it works and why it's useful.

3.2.1 CLOS Example—Instance Representation

Consider the way that instances (objects) are represented in an object system[3]. The representation is an implementation issue; however, the implementor's choice of representation will affect the way that implementation can be used, and the sorts of applications which it will efficiently support. The applications programmer has to hope that the implementor has chosen a representation which will be efficient for his particular applications.

If an applications programmer is writing a graphical system, it might well include a definition for the class of Cartesian points. Some of the properties of this class and its instances are that: there will likely be very many instances; each instance has only a few slots; each slot has a value for each instance (*i.e.* there are no empty slots); and that fast access to slot values is essential. These requirements suggest that the most appropriate representation of the instance is a "packed" representation, in which the instance representation is small (so that many can be created) and in which each slot value is stored at a fixed point in the instance (so that lookup is fast).

However, the applications programmer might also write a program in which the requirements are quite different. Consider a knowledge-based AI system, for instance, in which the programmer defines a class "person" to refer to people. Some of the properties of *this* class will be: there will be relatively few instances; each instance will have potentially very many slots; and a given instance will probably have only a few slots filled in. In this case, a different instance representation is desired, in which space is only allocated to slots when values are given for them (otherwise each instance will have a very high memory overhead). We could call this a "sparse" representation.

So, we have two different cases, each requiring a different instance representation. Usually, the designer of the object system would have to decide which implementation path to follow, and hence which form of application to support and which to reject. In CLOS, however, both of these representations, or any other which the user (applications programmer) wishes, can be specified through the use of the *meta-object protocol*. Meta-objects are the internal objects of implementation, which are available to the user (such as objects, classes and methods). The details of these objects, and the behaviours and interactions defined for them through the internal methods of CLOS, form the meta-object protocol. Since the meta-objects are also manipulable CLOS objects, the applications programmer can create specialisations of them with different behaviours using the standard mechanisms of object-oriented programming.

In this case, we could support both instance representations, as follows:

1. Instance creation and slot allocation are behaviours associated with classes, rather than instances. Let's say the default behaviour in a given implementation is to use the packed representation; so the programmer wants to change the system to use the sparse representation in certain cases.

2. While CLOS may have been standardised, its pronunciation has not. 'Kloss' and 'see-loss' are common variants.

3. In CLOS terminology, objects are called the *instances* of a class, and their data fields are called *slots*. Messages which can be sent to objects are called *generic functions*, and the specifications of their behaviour are called *methods*.

2. CLOS's default class behaviour, including the usual slot allocation mechanism, is specified by a class called `standard-class`. All classes are instances of standard-class (*i.e.* standard-class is their *meta-class*[4]), and they inherit class behaviours from it.
3. To create a new class behaviour, the applications programmer would create a new class as a subclass of standard-class; call this `sparse-class`. This is the class which will have the lazy slot allocation mechanism. Thus, we will use sparse-class as the meta-class for any class which is to have this mechanism. The `defclass` form, which defines new classes, lets the programmer specify a non-default meta-class.
4. The meta-object protocol specifies the generic functions which control instance creation, slot allocation and slot lookup. The programmer would define new methods for these functions, specialised for members of the class sparse-class.
5. Classes such as person can now be created as instances of sparse-class. When instances are created of an instance of sparse-class, the new programmer-supplied methods will be invoked as part of the instance creation mechanism. Similarly, slot lookup will cause the programmer's methods to be called. Thus, within the CLOS framework, we have developed an extension allowing a different slot allocation mechanism to be used.

3.2.2 Reflection and Object-Oriented Programming

The relation of reflection to object-oriented programming is crucial in this example, and it is the techniques and features of object-oriented programming that make it practicable to use reflection in this way in CLOS. In particular, two aspects of object-oriented programming are important:

1. *Inheritance* makes it easy to incrementally specify new behaviours. In the slot allocation example, new methods only had to be defined for three aspects of the operation of sparse-class. All other behaviours were inherited from the superclass of sparse-class, standard-class. Thus an incremental change like this merely involves specifying what behaviours differ from the default. All else is handled automatically through inheritance.
2. *Polymorphism* makes it possible for multiple implementations of the same process (*e.g.* slot allocation) to coexist. The appropriate methods will be called for a member of a given class, and may have different behaviours without any clash. This in our instance allocation example, both the packed and sparse representations could be used at the same time.

4. A meta-class is a class which describes classes. Thus a class has two relationships to other classes; it will sub-class (inherit from) its superclass, and, at the same time, is an instance of (responds to messages defined by) a meta-class.

3.3 Silica, A Reflective Window System

If we turn to Silica [Rao, 1991], our third example, we can see the use of reflection in what might be called an application domain. Silica is a generic window systems architecture, which embodies computational reflection; it forms the basis of the Common Lisp Interface Manager (CLIM). Unlike CLOS, which is implemented in CLOS, and 3-Lisp, which is implemented in 3-Lisp, a window system is not implemented in a window system; hence, reflection in Silica is not metacircular. However, the reflection is still very much present and, as in CLOS, it allows the programmer to examine or manipulate structures within the system itself—that is, a system can get involved in the implementation of a system that underlies this. Rao refers to this as *implementational reflection*.

We can consider an example very like the instance allocation example in CLOS. Consider the representation of a window in a window system. For some applications, a window is a "heavyweight" object—it has many properties, there are only a relatively small number on the screen at once, they are of variable size and may overlap. However, in a spreadsheet application, a more "lightweight" notion of a window might be appropriate. For example, if every cell in the spreadsheet were to be a window, we would wish to reduce the memory overheads, and to be able to take advantage of the fact that windows are now always a fixed sized and tiled rather than variable size and overlapping. With so many windows, we might wish that they only be created and stored as needed, and that all the data associated with a window not be stored when the window is not visible. In much the same way as CLOS allows programmers to subclass and change the implementation of object system structures, Silica allows them to change window systems structures, and so a more efficient window representation could be created for that particular application.

Once again, then, reflection has allowed the programmer to build an efficient implementation of a case which was not directly anticipated by the designer of, in this case, the window system. Thus Silica avoids a trade-off which is traditionally the domain of the system implementor, and instead allowed the user (a programmer using Silica to build a windowing application) to make the sensible choices for particular application situations.

4 Recap—Reflection in General

Having now looked at three examples of computational reflection in different systems, we are in a position to step back and look at reflection in general. In 3-Lisp, CLOS and Silica, there has been a general move to make systems malleable through the incorporation of the principles of reflection. This malleability allows the systems to be made compatible with other hardware, software and infrastructural systems; to be open-endedly extensible; and to be efficient in a range of different circumstances, even when those circumstances could never have been foreseen by the systems designer. The systems designer has been able to avoid the traditional trade-offs of systems development, and the potential for "premature commitment" to a restricted set of applications. All of these make a reflective architecture a good choice for flexible systems design.

Of course, there is a "down side". In particular, meta-object protocol design is hard, especially if a fully open-ended extension mechanism is to be realised. And, once the protocol has been designed and implemented, it can be difficult to use. Like any powerful tool, the potential for damage if it is abused is great. It would seem likely that such risks can be reduced through the use of appropriate programming tools, and in particular, tools which will provide information on the design of the meta-object protocol and the expected usage patterns. Design notation techniques such as QOC [MacLean et al, 1991] or tools which annotate code libraries with explanations of the design such as the Cognitive Browser [Green *et al*, 1992] could be ways of providing the programmer with the information needed to make appropriate application design decisions. When such decisions can be made effectively, we can use the full power of the principles of reflection to build flexible systems.

5 Reflective Architectures and CSCW

5.1 CSCW—The Problem of Flexibility

Now that the computational reflection model has been laid out, we can turn our attention to the design of systems for Computer-Supported Cooperative Work. Such systems are widely recognised as being hard to design well; and I would claim that one particular reason for this is the requirement for flexibility. Flexibility is a particularly critical issue in the design of CSCW systems precisely because it occurs at all levels, from the interface to the infrastructure.

We can identify three particular classes of flexibility required of CSCW systems.

1. *Static* flexibility is flexibility in the system which probably does not change, or will change only slowly, in the course of a collaboration. Examples of these are the various forms of tailorability, for individuals or groups; an accommodation of different working styles, which might be imposed by individuals, tasks or organisations; and the need to adapt to the requirements of other systems already in place (*e.g.* compatibility with existing single-user applications).
2. *Dynamic* flexibility involves responding to the changing aspects of the group. For instance, the group behaviour will change a great deal in the course of a collaboration, and those changes will be extremely fluid and tacit. In the course of a meeting, there may at different times be a concentration on a particular activity by all members of the group, or periods when individuals work privately on something to bring to the group later, and so on. Further, even the group membership itself may change; in a questionnaire study of collaborative writing, Beck reports that 22% of respondents reported that the membership of the group changed even after the writing phase had begun [Beck, 1992].
3. *Implementational* flexibility is the requirement that a system be able to adapt to a variety of hardware platforms and other network and system infrastructures. These can require not just minor changes, but major reorganisations in the structure of the system's activity.

The degree of flexibility required of CSCW systems, all across the board, poses major problems in their design. The usual solution is to constrain the applications being developed to particular situations, in which only a subset of the possible range of facilities need be implemented. However, in a CSCW toolkit, such constraints are a liability; we would not wish design choices in the implementation of the toolkit to overly constrain the user of that toolkit who wishes to build collaborative applications. Therefore, we need to find a way to deal with the flexibility.

5.2 Dealing with Flexibility through Reflection

We have already seen how computational reflection can help in the design of flexible, malleable and extensible systems. Since CSCW design is in need of such systems, it seems obvious to bring the two together. The reflective approach lets us attack the problems of flexible implementation over a range of infrastructures; it helps us separate issues of representation, presentation and manipulation, and so accommodate multiple perspectives, tailorings and so forth. In fact, reflective architectures are *particularly* suited to the design of CSCW systems, principally because of the problems of supporting dynamic flexibility. A reflective system embodies a model of its behaviour which is causally connected to that behaviour, and when such a model is available it becomes possible for a system to observe changes in the model and adapt dynamically to those changes. Not only does this dynamic aspect of reflective architecture lend itself particularly to CSCW, but overall, reflection provides a coherent framework for understanding the flexibility required throughout the system. The consistent and comprehensible model that it provides of flexibility throughout the system becomes a rationalising principle for making sense of changes and their impact.

We can contrast this approach with ongoing debates in the CSCW design community about particular elements in the design of CSCW systems. Some arguments centre around “centralised” versus “distributed” approaches; others ask whether “tightly-coupled” or “loosely-coupled” interactions are better; while others debate which particular work style ought to be supported in a tool (since there can only be one). From the reflection perspective, these are all implementation decisions which the toolkit designer cannot afford to take; and indeed, it is unclear whether the developer of a particular application is in a position to make the decisions and implement the application based on only one approach. Reflection lets us manage all of these, and adapt dynamically to the requirement of individuals, groups and organisations throughout their collaborations.

6 A Meta-Object Protocol for CSCW

6.1 What Would It Look Like?

The talk of using reflection in CSCW design begs the question, “What would a meta-object protocol for CSCW look like?” At this stage, I’m far from being able to answer that question, although I can outline aspects of an eventual answer. We are considering a reflective toolkit for building CSCW applications; the need for reflection (or rather, the need for flexibility which it might supply) goes deeper than the interface, but not necessarily as far as the programming language itself. What does this toolkit generalise over? Answers include such things as implementation strategies for network connections, locking and access strategies for shared objects, issues of presentation of information and interface questions.

The design of the meta-object protocol, then, is the hard research question, and will take much work. A first approach is to make some suggestions about its structure, and then evaluate them by determining to what extent they can be used to describe existing, documented systems, and to what extent they provide sufficient flexibility to build interesting new ones. This is the process in which I’m currently engaged. A model of the meta-object protocol, then, might include as meta-objects:

1. *Shared objects*, which are the objects manipulated in the shared space created by the CSCW application. We would wish to be able to deal with the objects generally independent of particular edit requirements and the differences between, for instance, graphical and textual objects. Appealing again to questions of presentation, it might be that the same object has both graphical and textual forms in a given application, and so a more general description of “shared object” properties and behaviour is necessary.
2. *Users*, referring to the individuals who use the system, carrying with them particular tailorability preferences, access rites, roles and so forth. Groups are formed as collections of users.
3. *Sites*, which are the locations where users and objects may reside. The linkage between sites and objects, for instance, may determine distribution and replication patterns for shared information.
4. *Invocations/sessions*, referring to group activities which should be taken together as a group.

This is a very tentative model, and I would in no way wish to claim any particular validity for it. However, it gives a flavour, perhaps, of what the meta-objects may be in a reflective CSCW toolkit. From these objects, we would wish to be able to define behaviours such as the distribution polices, locking and access policies for shared objects, subgrouping, mechanisms for tailoring information and presentation styles, and so on.

6.2 CSCW Example—Object Locking

To show how a CSCW meta-object protocol can be applied to the design of flexible collaborative systems, let’s take the example of object locking in shared workspace systems.

All shared workspace systems must include a mechanism for locking access to objects—systems with no locks are merely the degenerate case. Like the instance representation in an object system, the choice of locking strategy is made by the system designer (in our case, the designer of a toolkit). The strategy chosen has implications for the applications which can be built with the toolkit. For instance, the no-lock strategy may be fine for a simple shared whiteboard program used by small groups of designers, but will be unsuitable for an application where data integrity is critical, such as group software development, or the management of precise engineering drawings. Similarly, a “strong” locking model, where locks must be explicitly requested and granted by users, which is appropriate for the software development case, will restrict the utility of the shared whiteboard program.

The outline of the approach taken by a meta-object protocol specification is doubtless familiar by now. We define the mechanism which underlies the locking strategy. Essentially, locking is a behaviour which relates users to shared objects. Thus:

1. In the meta-object protocol, we might define a generic function `obtain-lock` which specialises on users and objects. With this, we can define different locking actions for different combinations of users and objects.
2. We can create subclasses of the shared object class called `strong-lock-object` and `null-lock-object`. We can then define methods for the obtain-lock function for each of these classes, with different behaviours. Objects of class `strong-lock-object` will have an explicit grant/request behaviour; objects of class `null-lock-object` will have no locking (*i.e.* a lock request will succeed immediately).
3. Similarly, we can create subclasses of the user class with different lock behaviours. These might correspond to the roles the users play at any given point in the collaboration. For instance, we could define methods so that `obtain-lock` will always fail for users who are "readers" but not "writers", so that they cannot obtain the lock they need to change an object.

So, rather than embodying the locking strategy in the toolkit so that only one (or a predefined number) can be used, we have defined the mechanism by which the locking strategy is determined, so that any number of possible locking strategies can be used. Further, different strategies can be used for different objects which exist within the same shared workspace, or for different users, or at different times in the course of a collaboration. By defining the behaviour of the CSCW toolkit using a meta-object protocol, we have achieved the support for dynamic flexibility which was posited as a requirement.

6.3 Using the Meta-Object Protocol: ShareITs

Even at this early stage, before we are in a position to use the meta-object protocol as the basis of an implementation, we can certainly begin to use it as an organising principle for a study of collaborative tools. Indeed, this is the tack which we are beginning to take in work on ShareITs [Bellotti et al, 1991]. The first stage of this research has involved the development of a design space for collaborative tools using the QOC notation. A reflective model of implementation can be used as a way of understanding the relationships between the components in the design space. A system which implemented every option would be very flexible indeed (if not terribly coherent, perhaps) and, as suggested earlier from an implementational perspective, the notion of reflection provides a framework in which such flexibility can become manageable. This is in accordance with our goal of trying to find the principles behind a generic architecture for the design and implementation of shared tools.

7 Summary

In this paper, I have attempted to outline the notion of computational reflection and show how it applies to the domain of CSCW applications.

Computational reflection provides a model by which, through the use of a causally-connected representation of its own behaviour, that system can achieve an open-ended flexibility. Integration of reflection with the techniques of object-oriented programming allows its power to be made available conveniently in computational systems.

The application of computational reflection to CSCW design allows us to tackle the fundamental problem of the need for flexibility in CSCW systems; this flexibility requirement appears on all levels of the design. Further, the reflective model is particularly applicable to CSCW design because of the problem of *dynamic* flexibility in such systems, which can be tackled through introspection on a representation of the system's behaviour.

This work, then, is part of a general trend emerging within the reflection community to take the model they have developed and apply it more widely to other areas of computer science. Indeed, it seems that reflection as a design approach may extend beyond that and have applications to the wider "systems" design field.

Acknowledgements

The ideas expressed in this paper arose from discussions, both face-to-face and imaginary, with a number of people. I would especially like to thank Victoria Bellotti, Doug Cutting, Wendy Mackay and Eliot Miranda for invaluable contributions to their development. Eevi Beck and Abi Sellen also provided especially useful comments on earlier drafts, without which this paper would surely be totally incomprehensible.

References

[Allen, 1978] John Allen, "*Anatomy of Lisp*", McGraw-Hill, New York, 1978.

[Beck, 1992] Eevi E. Beck, "*A Survey of Experiences of Co-authoring*", in Sharples (ed.), "Computer Supported Collaborative Writing", Springer-Verlag, forthcoming.

[Bellotti et al, 1991] Victoria Bellotti, Paul Dourish and Allan Maclean, "*From Users' Themes to Designers' DReams*", EuroPARC Technical Report EPC-91-112, 1991.

[Clinger and Rees, 1986] Will Clinger and Jonathan Rees (eds.) ,"*The Revised^3 Report on the Algorithmic Language Scheme*", ACM SIGPLAN Notices, December 1986.

[des Rivières and Smith, 1984] Jim des Rivières and Brian Smith, "*The Implementation of Procedurally Reflective Languages*", Xerox PARC Report ISL-4, June 1984.

[Green *et al*, 1992] T.R.G. Green, D.J. Gilmore, B.B. Blumenthal, S. Davies and R. Winder, "*Towards a cognitive browser for OOPS*", Intl. Journal on Human-Computer Interaction, 4(1), pp 1–34, 1992.

[Kizcales et al, 1992] Gregor Kiczales, Jim des Rivières and Daniel Bobrow, "*The Art of the Meta-Object Protocol*", MIT Press, 1991.

[Maclean et al, 1991] Allan MacLean, Richard Young, Victoria Bellotti and Thomas Moran, "Questions, Options and Criteria: Elements of Design Space Analysis", Human-Computer Interaction, 6 (3&4), pp 201–250, 1991.

[McCarthy et al, 1960] John McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park and S. Russel, "*Lisp 1 Programmer's Manual*", Artificial Intelligence Group, MIT, Cambridge, Mass., 1960.

[Smith, 1982] Brian Smith, "*Reflection and Semantics in a Procedural Language*", MIT Laboratory for Computer Science Report MIT-TR-272, 1982.

[Rao, 1991] Ramana Rao,"*Implementational Reflection in Silica*", Proc. ECOOP 91, Geneva, Switzerland, July 1991

[X3J13, 1988] Daniel Bobrow, Linda Demichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales and David Moon, "*Common Lisp Object System Specification*", X3J13 Document 88-002R, June 1988.