Pragmatic Formal Design:

A Case Study in Integrating Formal Methods into the HCI Development Cycle

Meurig Sage and Chris Johnson,

GIST,

Department of Computing Science,

University of Glasgow, Glasgow,

Scotland United Kingdom, G12 8QQ.

E-mail: {meurig,johnson}@dcs.gla.ac.uk

Abstract Formal modelling, in interactive system design, has received considerably less real use than might have been hoped. Heavy weight formal methods can be expensive to use, with poor coverage of the design life cycle. In this paper, we suggest a pragmatic approach to formal design. We rely on a range of models that can help at different stages of development. We use as a case study, the design of a multi-user, design rationale editor. In the initial stages of our design, we use a range of semi-formal notations, to perform a task analysis. We then develop a prototype in Clock, a high level functional language. From this, we derive a LOTOS specification, which we use to verify that our system satisfies important design requirements. The task analysis helps here, in highlighting these requirements. Throughout we rely on tool support to simplify the process, and so make it cost effective.

1 Introduction

Formal specifications, of interactive systems, have been proposed as a means of improving design [Harrison1994, Paterno1995]. Unfortunately, they have experienced less real use than might have been hoped. Fields et al [1997b] argue that they can be overly cumbersome for use in initial design stages. Developers often do not wish to expend the considerable amount of effort necessary in producing detailed formal specifications, if they may need to reconsider the whole design and alter significant portions of the model. It can be difficult to get a good idea of what a system will look like without an initial prototype. Certainly designers may be unwilling to carry out complex proofs until they are reasonably happy with the initial design. Instead, Fields et al [1997a] argue that lightweight, simple models may be useful to help focus in on specific design problems. Only later might a more detailed specification be useful to help guarantee properties about a system, and only then if it could be produced in a cost efficient way.

1. A Pragmatic Design Approach

This paper presents a design method that deals with some of these problems. We take a pragmatic approach that aims to:

- re-use existing techniques;
- make good use of tool support to assist design stages;
- provide good coverage for all stages of development.

A summary of our approach can be seen in Figure 1. We use a set of lightweight models at the initial stages of the design, including high-level task designs, paper prototypes and more detailed interaction models to focus on parts of the design. These allow us to quickly focus on early design issues. We develop a prototype in Clock, a constraint based language [Graham1996a]. This allows us to program at a very high level of abstraction, and enables us to quickly produce a well structured prototype. We then derive a LOTOS specification from this prototype. This allows us to develop a full specification of the system with a minimum of cost and effort. We can then attempt to guarantee properties about the system using LOTOS simulation [Pavon1993, Markopoulos1997] and model checking [Garavel1996, Paterno1995] technology. We rely here on the earlier task analysis to direct the formal model checking. We make use of design rationale methods [Maclean1996], to attempt to justify design decisions throughout, and to link parts of the design together. The whole design method is iterative; each stage can feed back into earlier stages. We do not formalise each and every step, but we do maximise our use of well developed HCI techniques such as task analysis [Johnson1993], design rationale and iterative development. This pragmatic approach is adopted because of the costs associated with full formal refinement.

The paper is organised as follows. In Section 2 we introduce our example, showing why it is appropriate. We then introduce each stage of our design method in Section 3. In Section 4 we then discuss the actual design, highlighting some important development decisions that were made, and how each stage helped us to consider these decisions. Finally, in Section 5 we present some conclusions about the effectiveness of our approach.

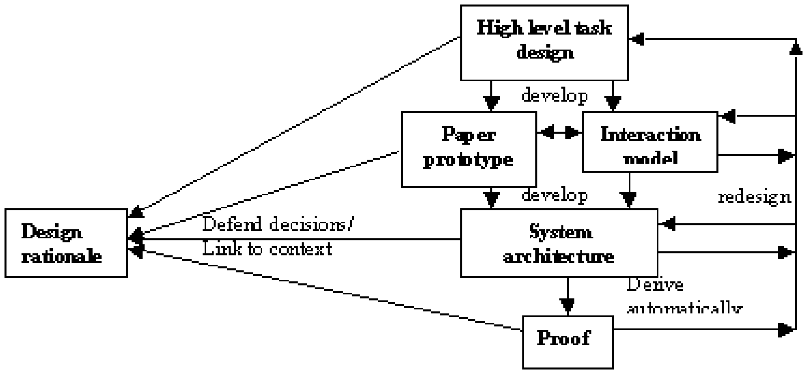

Figure 1 - The Design Method

2 Case Study: A Multi-user Design rationale Editor

2.1 The Case Study

We use, for our example, the design of a multiple user, design rationale editor. This provides for a suitably complex case study, where significant design decisions are needed and where concurrency is required. This concurrency adds an extra level of complexity that makes formal specification and proof helpful. For instance, in ensuring liveness and safety requirements are satisfied for multiple users of a shared workspace.

2.2 Design rationale

Design rationale has received a lot of attention [Moran1996]. A number of semi-formal notations have been developed that attempt to document clearly why design decisions were made. The Questions, Options and Criteria (QOC) notation is one such notation developed at Rank Xerox [Shum1991]. It is a graphical notation that highlights key questions in a design, and links them with possible options and criteria that support those options. Several studies [eg Shum1993] have highlighted the need for tool support for this notation. A variety of tools have been developed, frequently based on hypertext systems. However, current tool support is frequently inadequate for designers needs [Buckingham Shum 1996]. In particular, tools often provide little support for multi-user activities. Buckingham Shum [1996] argues that design rationale itself is still in its infancy. This makes it difficult to be sure exactly how designers will wish to use these tools. Iterative development is therefore required to explore possible designs that will satisfy the needs of designers.

2.3 Collaborative Software

This case study is appropriate because the development of collaborative software of this form is still in its infancy. It is frequently difficult to determine exactly how a group of users may wish to collaborate using a piece of software. It is very easy to produce software that does not properly consider how a group of users may share a design, and so seriously hinder the use of such a tool [Bentley1994]. A design therefore needs to be well thought out, and will frequently go through several iterations before it can be useful. It can also be difficult to produce software for several users because of the concurrency involved. Complex locking mechanisms may be required that need serious thought [Dix1992]. A design therefore needs to be well structured.

2.4 Our Prototype System

Figure 2 shows the interface for the prototype system we produced. It allows several users to build a QOC rationale. Users can create nodes (Questions, Options or Criteria) and connect them together. They can edit the design at any time. When one user is modifying a node, other users are prevented from manipulating it. Users can also note their actions, and any extra textual information using a shared log. Finally, each user can hide or view certain nodes to make browsing easier. Using the view menu, they can filter their view of nodes to show only Questions and Decisions; Questions, Decisions and Criteria; Questions and all Options; Questions, all Options and Criteria. They can also hide all nodes following a particular node, as has been done with "Q:24" in Figure 2. There are problems with our initial interface design. However, these will be removed through the process of iterative prototyping and evaluation.

We will now discuss our design method, and how each design phase helped in the development of this system.

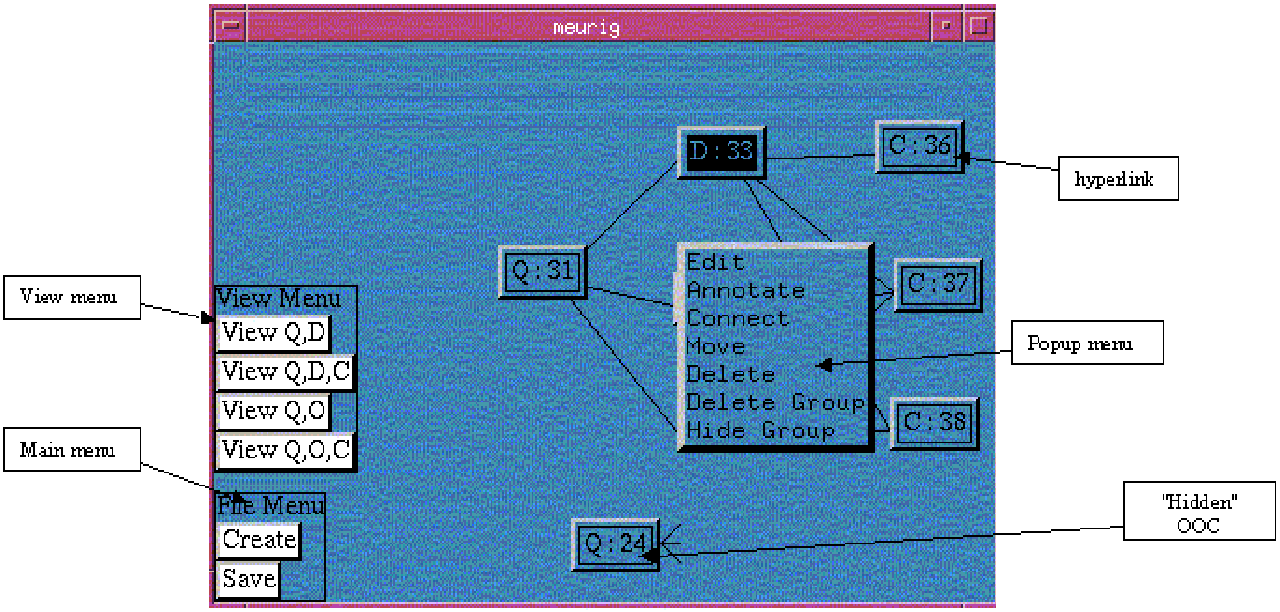

Figure 2 - The QOC editor

3 The design method

3.1 Requirements Elicitation

As discussed earlier (Figure 1), we use an iterative development method with several stages. The very first stage in the development process is requirements elicitation. We need to gain contextual knowledge, from activities such as observational studies and questionnaires. For our example, we rely on the results of observational studies of design rationale use by Buckingham Shum and Maclean [Shum1993, Buckingham Shum 1996, Maclean 1996]. We concentrate on the design approach that follows these initial requirements gathering exercises.

3.2 High level task analysis

We use a high-level task design is to summarise the results of a contextual analysis, and structure the design. This is accompanied by initial paper prototyping to aid in focussing the design. More

detailed specifications and prototyping can follow. Design decisions at each of these stages are linked back to the contextual information that has been gathered, through the use of design rationale. This process is, however, very iterative. For instance, a basic amount of prototyping was required to gain a deep enough understanding for a good high-level task specification.

3.3 Semi-Formal Interaction Design

When developing complex concurrent systems, interaction becomes more complex. This means that designers should be able to consider and discuss the details of multi-user interaction. Scenario design can help in describing how users may interact with the system and each other [Clark1997]. Semi-formal notations such as the User Action Notation (UAN) [Hartson1990] help designers to do this by focussing on particular tasks. Producing a complete UAN specification for a design can be costly. However, when used in a more focussed way it can be helpful, for instance, when considering the complex locking mechanisms in a collaborative system. UAN has the major advantage that it is a relatively simple notation to learn. A study by Johnson, comparing UAN with temporal logic and transition networks, showed that designers found it relatively easy to read [Johnson1997]. One important use of UAN here can simply be documenting design arguments, so they are visible to others. Informal reasoning is also supported. The obvious relationship between user actions and interface feedback makes it easy to consider issues such as the predictability of a design. However, UAN has some problems, due to its lack of formal semantics. It can be difficult to express complex actions that require choice. However, this can be overcome, where necessary, by using LOTOS temporal operators with it, to describe the more complex activities. XUAN [Gray1994], an extension to UAN, made use of LOTOS like operators. As we show in Section 4.5, the focus provided by UAN can be used as a basis for testing the system design later on.

3.4 Prototyping

The next important stage in the design is prototyping. Recent tools for rapid prototyping, such as Visual Basic, have made developing software easier. A simple prototype can be developed very quickly. However, such tools tend to sacrifice structure for flexibility [Graham 1996]. It may be easy to produce a piece of software, but if it becomes unmanageably complex it may hinder iterative development. This becomes a more significant problem when dealing with multi-user systems. The development of such systems requires concurrency. Some data may be shared between a group of users; other data may only be accessible to one user. This sort of development stretches many first generation HCI prototyping tools such as Alexander's "me too" [Alexander1990]. It requires some structure to make it clear to a programmer how data is shared, and what effect several users acting on the same data will have. A high level, well structured development language is therefore needed. Clock [Graham1996a] provides such a language. It

provides a high-level, constraint based, programming language. Architectures are built visually, in a component based way, giving a good understanding of the structure of a system . The Clock language has much in common with the York interactor [Harrison1994] style of specification. We will describe Clock and discuss this comparison in more detail in Section 4.3. Graham et al [1996b] suggest a method for developing UAN specifications into Clock architectures. This development can therefore take place in a structured way, and we can develop prototypes rapidly.

3.5 Formal specification and Proof

When building a complex concurrent system it can be difficult to guarantee its behaviour. Though structured design and semi-formal models are helpful, formal specifications and proof can be very important. For example, though UAN supports informal reasoning, it does not support interaction proofs. Recent work on formal system modelling can help here. However, designers may be put off if the effort required to develop a specification, and prove it correct, is too great. To make this stage as simple as possible, we derive a LOTOS interactor network from our Clock prototype. When we are satisfied with the initial design, the Clock architecture can be transformed into a LOTOS specification (See Section 4.4). We can then perform some proofs with this specification (See Section 4.5). We can also demonstrate task conformance between the LOTOS behaviour of the UAN specification, and the LOTOS system specification [Markopoulos1997].

4 The QOC Editor Design

4.1 High level task specification

The first stage in the design was the development of a high-level task specification. For this we made use of ConcurTaskTrees [Paterno1997]. This notation provides a graphical representation, in a tree structure. Tasks can be related by a set of operators, based on those provided with LOTOS. The notation separates tasks into four different categories, user tasks, application tasks, interaction tasks and abstract tasks. User tasks are those carried out only by the user or between groups of users; application tasks are those carried out completely by the system; interaction tasks involve the use and the application, while abstract tasks relate to higher level goals in the system.

The LOTOS operators that we use, in this example, to compose tasks together are:

- interleaving (|||) - two tasks can be carried out in any order
- sequencing (>>) - two tasks are carried out one after another
- choice ([]) - either of two tasks can be performed

The task specification for our design is shown in Figure 3. It says that the tasks involved in use of the QOC editor consists of creating and redesigning QOCs. Creating a QOC consists of adding questions, options and criteria (*Create Nodes*) and noting down any more detailed information about these nodes (Support Rationale). These nodes will then be connected together (*Connect Nodes*). For instance, supporting criteria will be linked to objects. The QOC will then undergo discussion and redesign (*Redesign*). When dealing with a large design, users will probably wish to focus in on particular QOCs (*Focus*). Users will then modify their design. They may add new criteria, or delete old ones that are considered irrelevant. Particular nodes (Questions, Options or Criteria) may, for instance, be renamed (*Change Label*). Certain criteria may also be changed into options, or questions may be decomposed further (*Change Node Type*). These different activities have been noted in empirical and observational investigations into use of design rationale [Buckingham Shum 1996]. Users will also want to move objects around as a design continues to make the QOC more visually appealing (*Beautify QOC*).

The activity of focussing in on a design (*Focus*) may require different forms of support. Users may wish to follow a design through a set of questions and decisions, ignoring discarded options. They may also wish to see how different criteria relate to each other, or how the design has progressed over time [Buckingham Shum 1993].

Some tasks may simply be performed by individual users. However, there is a core of individual tasks that are dedicated to multiple user activity. The discussion activities (surrounded by a box in Figure 3) will involve the activities of multiple users. As the development continues users may also wish to work together to modify and create new design rationales.

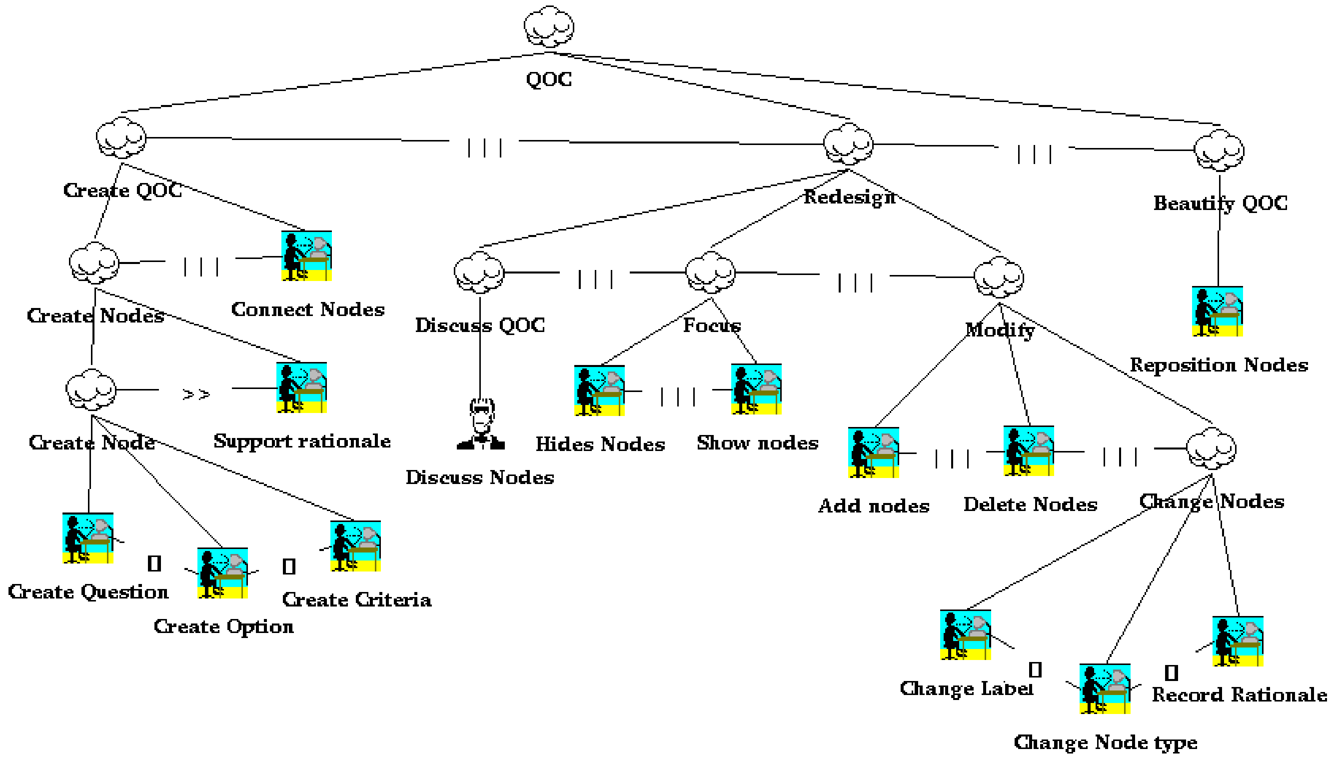

Figure 3 - ConcurTaskTree Specification

The use of a ConcurTaskTree specification provides a visually appealing, clear description of the basic aims of the design. The use of LOTOS operators gives a clear definition to the task design , and helps to maintain consistency across our design approach. Again, it also satisfies one of our prime objectives: to integrate existing HCI techniques to support formal methods in the development cycle.

4.2 Detailed Interaction Design

4.2.1 The Approach

The next stage in the development was to produce a more detailed system design. To do this we used a mixture of paper prototyping, UAN specifications and design space analysis. For the design space analysis, we bootstrap the design of the QOC editor using QOCs themselves. The use of QOCs here helped relate possible designs back to the contextual requirements. It also helped link information gained from UAN analysis to design decisions. This process was therefore one of literate specification [Johnson 1996], where different elements of a design are linked together, to make understanding it easier. For this stage, we could have made use of other design rationale languages, such as gIBIS [Conklin1989]. This use of QOCs again relates back to our pragmatic criteria as the notation is well known, simple and easy to read.

The paper prototyping was used initially to feed back into the task design. The key design feature needed was flexibility: designers should be able to put QOCs together in any order. Design tools have often faced problems because they restricted the way a designer could work. In our system nodes can be created and moved, connected or deleted in almost any order. Their text can be annotated, both the label and the type of the node (eg is it a Question , Option or Criteria). Users can also filter their view of the whole QOC. The resulting interaction is fairly complex, but becomes more so when several users are involved. We will consider the basic locking mechanism, and an example of users deleting parts of a QOC, as examples of how semi-formal notations helped our design.

4.2.2 Interaction Design Example I - Locking mechanisms

Users have to be able to design QOCs together. Though users may face no problems when working on different QOCs, when they come to discuss the design of one particular QOC in more detail they may face problems. At this stage, one user may attempt to edit a question, while another attempts to delete that question. Collaborative working can then become difficult.

There are several solutions that can be taken to this problem. The first is simply ignore it, and let users develop their own social protocols [Dix1992]. In this case, it is simply up to the users to agree a strategy that prevents them from modifying the same data simultaneously.

Alternatively, we could implement a locking mechanism that prevents more than one user altering the system at a time. Locking mechanisms can be implemented at various different grains [Dix1992]. For instance, we could force only one user to make any modifications to the QOC design at a time, thereby using a *coarse* grained locking mechanism. This would usually be too severe. A more sensible approach would be to prevent more than one user from modifying any node at a time. This results in a more *finely* grained locking mechanism.

Once we have decided on this general mechanism, we need to consider exactly how the locking mechanism should be implemented for a particular interface. For instance, we should consider when exactly should locks be enabled: should the locking mechanism be implicit or explicit.

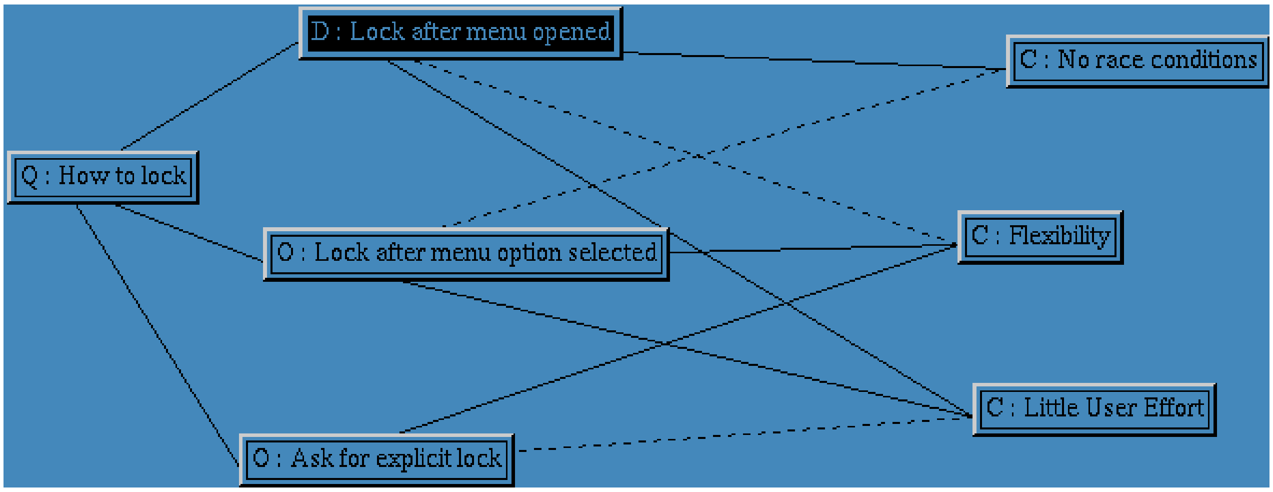

Figure 4 - QOC for basic locking mechanism

The QOC in Figure 4 can be used to summarise the argument. Implicit locks require less effort on the part of the user and are therefore to be preferred. The possible race conditions mentioned in the QOC criteria can be understood by looking at the following UAN specification (Figure 5). Two users both attempt to alter a node. Both have accessed the menu, but user two clicks faster than user one. User one may therefore be confused that they have attempted to modify a node but have failed part way through. Another possibility would be to lock the node immediately after the menu was selected. This would prevent this sort of race condition. However, it would also prevent one user accessing the node's log while another user was editing it. UAN helps highlight this sort of scenario in an easy to read way. It helps us consider this kind of interaction and guarantee that we provide appropriate feedback in cases of mutual interaction. It makes it easier to share these considerations and decisions with other designers. This would be very much more difficult with the ConcurTaskTree (Figure 3) representation because of its higher level of abstraction.

User 1 User 2

User Action	Feedback	Interface State	User Action	Feedback	Interface State
Select menu	Menu appears				
			Select menu	Menu appears	
		Node appears locked on user's screen	Select edit	Unlocked => Enter edit mode	Unlocked => Lock Begin Editing
Select delete	Locked => Error message	Do nothing			

Figure 5 - UAN Spec - Lock after menu selection

4.2.3 Interaction Design Example II - Hiding and deleting nodes

A potentially serious problem can be discovered when we look at filtered views of a QOC design. This problem emerged as a result of our UAN analysis, and so provides a good example of why the notation is useful.

Users need to be able to hide part of a QOC design when it becomes large enough, so as to prevent too much clutter on the screen. In our editor, we provide a number of hiding strategies. One of the above is the ability to hide part of a QOC. For instance, consider the situation in Figure 2. There are two QOCs in existence, but only one of them is fully viewable. This ability to hide QOCs has an effect on the ability to treat them as groups, particularly the ability to delete. We might wish, for example, to be able to delete a group of QOC nodes. Several possible choices are shown in Figure 6.

We can choose not to allow deleting of objects when they are filtered. Alternatively, we can choose to reshow the whole group and then delete that node. The most flexible alternative is to provide a "Delete Group" operation. This flexibility is important as it relates to what Green [1989] calls viscosity. Without it, making changes to the QOC editor would be far more cumbersome.

However, when adding this flexibility, we have to seriously consider whether it will be safe, in relation to multi-user interaction.

We can demonstrate a possible problem and solution using UAN (Figure 7). If user one attempts to delete a whole QOC, while user two is altering part of that QOC, there will be a problem. This will require some form of feedback. Again UAN can be used to consider the scenario and attempt to provide a safe route through it

.

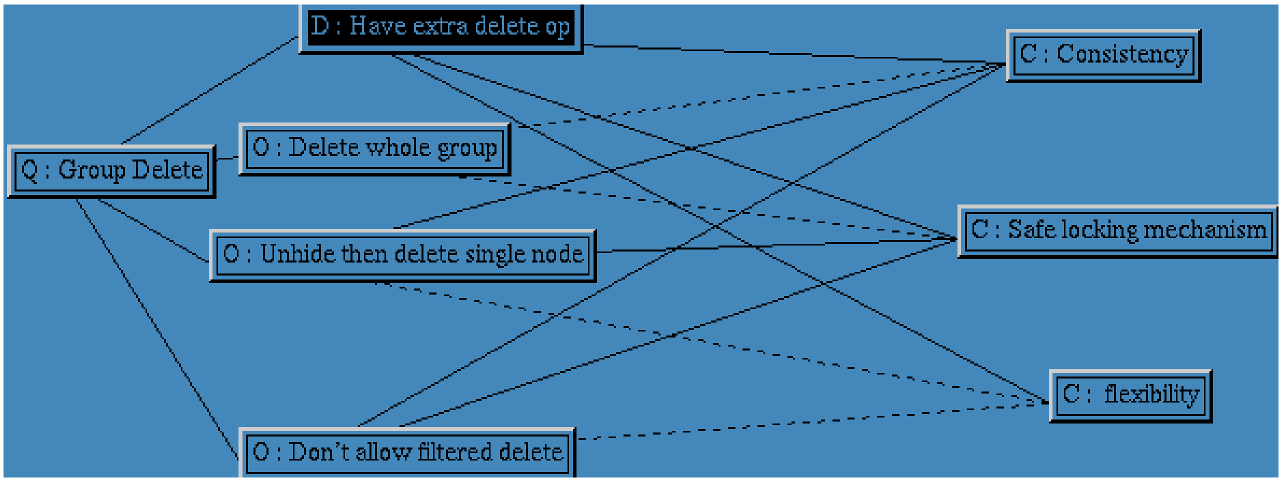

Figure 6 - QOC to discuss deleting

At an early design stage, a number of problems will be faced. Designers need to be able to consider such activities without resorting to heavy weight formal models. They are, in fact, unlikely to want to do so. Instead, more task centred, light weight models are easier for considering the initial interaction design. A variety of modelling approaches can instead be used [Fields1997a]. Though UAN is a semi-formal modelling language, with many restrictions [Harrison1994], it can still be used to successfully consider early design problems.

User 1 User 2

User Action	Feedback	Interface State	User Action	Feedback	Interface State
			Select menu for Question Q	Menu appears	
			Select Hide Group after Q	All nodes after Q vanish (including criterion C)	
Select menu for criterion C	Menu appears				
Select edit	Unlocked => Enter edit mode	Unlocked => Lock criterion C			Criterion C locked
			Select menu for Question Q	Menu appears	
			Select Delete Group	Not all unlocked => error message *	Not all unlocked => do nothing

Figure 7 - UAN for delete group design

4.3 Developing a prototype

4.3.1 Possible Approaches

While we can use design notations to consider the interaction in a system, to get a real idea of what a system will look like we need to use a prototype. To do this, we need a high-level, rapid prototyping environment. This should minimise the translation between prototype and formal specification [Alexander1990]. There are a number of possible languages that could be used at this stage [eg Myers1990, Sage1997]. We make use of Clock a constraint based functional language [Graham1996a].

4.3.2 The Clock Development Language

Clock has a graphical architecture language that can be used to describe how systems fit together and a textual language to describe the behaviour of each component. It is based on the MVC [Krasner1988] model. A system is described by decomposing a design from the root view into a number of component sub-views. Components can communicate via constraints, and update events. A system therefore appears as a hierarchy of components.

Components contain an *event handler*, which takes user inputs and sends updates. This *event handler* is similar to the *controller* part of the MVC model. Components also contain abstract data types (*ADTs*) which represent the MVC *model* and can receive inputs and accept requests. Finally, a component has a view, which is defined as a relation of the model. This is therefore similar to Hill's ALV model [Hill1992].

The Clock approach also has much in common with the York interactor model and state based system modelling [Harrison1994]. Clock ADTs can be seen as similar to the internal state of a York interactor, the view function is like the York rendering relation, and the event handler is similar to a set of behavioural equations. The use of constraints to relate components is significant here. We can specify complex systems in terms of these constraints and so easily guarantee such properties as state-display conformance. The Clock architecture design makes it easy to consider such concepts as the visibility of a system, as we can define in the relation exactly what is visible.

Clock architectures are produced interactively using a visual tool called *ClockWorks* [Graham1996a]. This allows programmers to design and modify their architectures easily. It provides easy access to a library of components. Fast iterative design is therefore possible.

Clock also supports groupware applications [Graham1996a]. View functions are provided to allow windows to be opened up on different users screens. A distributed version has also been developed. Programmers can specify that some data should exist on the user s side and other data should exist on the client s side. This aids the easy development of groupware applications.

4.3.3 An Example with Clock

As an example of a simple Clock application, consider the definition of a button. When the user presses it, it appears pressed; when the button is released it performs an action. The Clock architecture for this object can be seen in Figure 8. It is made of three components, named *Depressed*, *ButtonStatus* and *ButtonView*.

The *Depressed* component is an ADT. It has three methods: *release*, *depress* and *isDepressed*. The first two are updates that set the state of the ADT, the last is a request method that returns the state. The *ButtonStatus* ADT is similar.

The *button* component is of class *ButtonView*. It accepts mouse actions (*mouseButton*, *enter* and *leave*). It can uses the methods provided by *Depressed* and *ButtonStatus*. It can also send a *buttonClick* event.

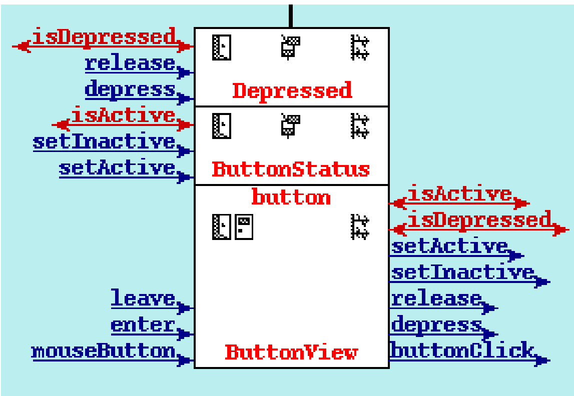

Figure 8 - A Clock Button

The controller part of the button component is defined as follows:

mouseButton "Down" =

if **isActive** then **depress**

else **noUpdate**

end if.

mouseButton "Up" =

if **isActive** then all [**release**, **buttonClick** myId]

else **release**

end if.

enter = **setActive**.

leave = **setInactive**.

When the mouse enters the button, the button becomes active; when the mouse leaves the button it becomes inactive. If the mouse button is pressed when the mouse is over the button it becomes depressed, if it is released over the button then a button click has occurred.

The view part of the button component is defined as follows:

```
view = let
setRelief v =
if isDepressed and isActive then
Relief "sunken" v
else
Relief "raised" v
end if,
buttonView = Text myId
in
setRelief buttonView
end let.
```

The button has a label, which is its name. When the button is both depressed and active, it appears in the sunken state. The view is therefore a constraint of the model, and simply specifies a relation between the model and the user interface appearance.

The complete code for the *Depressed* ADT is:

```
type State = Bool.

depress = save True.
release = save False.

isDepressed = this.
initially = save False.
```

The predefined function *save*, sets the new state of the ADT. The predefined function *this* returns the current state of the ADT. Therefore the *depress* and *release* methods simply set the state of the ADT to True and False, respectively. The predefined function *initially* says what to do when an instance of the ADT is created.

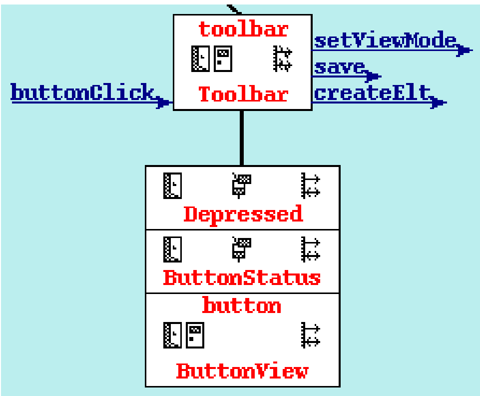

Figure 9 - Clock architecture for counter

There is a strict set of rules defined to explain how these updates can be used. Updates can only travel up the tree hierarchy. They are therefore guaranteed to terminate at or before the root. Infinite constraint loops are therefore impossible. Requests can also only travel up the tree, so a component can only use constraints based on values in its parent components ADTs. A component can use 0 or more instances of each of its subcomponents. These are created through the subview relationship. An instance of a subcomponent will be created when a subview is used to create that components view. For instance, we could create the toolbar of buttons, used in our editor (see Figure 2) as shown in Figure 9.

The toolbar view would be defined as:

view = above (map button labels).

labels = ["View Q,D","View Q,D,C","View Q,O",

"View Q,O,C","Create","Save"].

In this case six instances of the button sub-component would be created. These appear stacked vertically on the screen.

The Clock architecture language therefore provides good support for iterative design [Graham1996a]. Requests and updates from a component are routed automatically by Clock. Programmers do not explicitly say how components are to be connected. The only explicitly defined relationship is the subview. Requests and updates are simply routed up the tree to the first component that can deal with that type of action. This means that it is easy to move objects around in the tree, as no explicit connections will be broken. For instance, if we wished an abstract data component (ADT) that had been used by one component, to be shared by several, we could just move it up the tree. This would not cause any problems.

4.3.4 The Clock Design of the QOC Editor

The Clock architecture for this example is shown in Figure 10. The architecture structure relates closely to the visual appearance of the system. Each user has an editor view (*Editor*) which consists of the workspace and the tool bar (which has the save, create and view buttons.) The workspace consists of a set of nodes (Questions, Options and Criteria) along with connecting edges and possibly a rubber band line (if two nodes are being connected.) A node has a label, a local editor (*Annotations*) for the user and a view of the shared log for that object (*SavedAnnotation*). An edge will also have a log, a view and possibly a label. The system is built from a set of relations. Some data is local to individual users, some is global to all users. For instance, at the root level are ADTs that hold information about all the nodes, edges, positions and the shared log. At the editor level the list of nodes will be filtered, to consist of those that are visible because of the current view. The workspace component displays a node for every node visible provided by the editor.

The structure of the Clock architecture makes it easy to design and iteratively build complex systems. As an example of the ease with which modification can be made, we were able to add the different view filters (Questions and Decisions, Questions Decisions and Criteria etc) to the original design in under 20 minutes, in only a few simple lines of code.

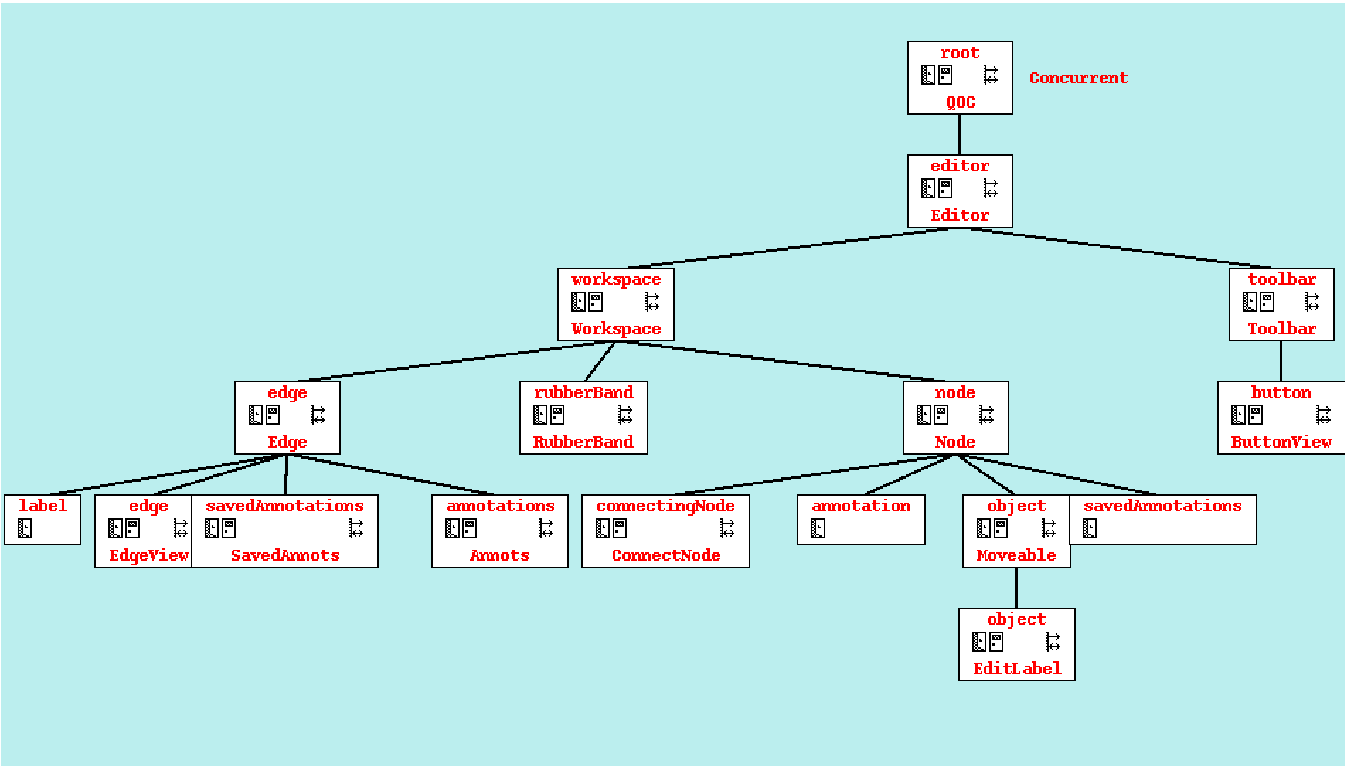

Figure 10 - Clock Architecture for QOC Editor Example

4.4 Developing a formal specification

The final stage in the development is to attempt to prove certain properties about our design. To perform interaction proofs, we could exploit an important relationship between Clock architectures and interactor networks. We can consider each Clock component to be equivalent to a LOTOS interactor [Paterno1994]. It accepts updates from the user side, and receives values from the application side via requests.

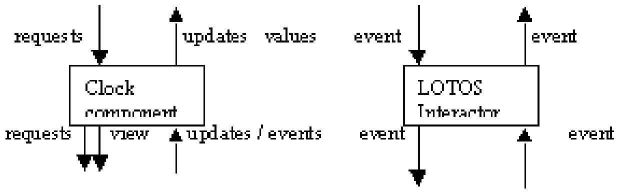

Figure 11 - Relationship between Clock and Lotos Interactors

We can translate a Clock hierarchy into an interactor network in three stages.

- **Turn each Clock component into an interactor**
- **For each subview relationship a parent component can send disable and enable events to its children.**
- **Link requests and updates between components.**

The significant difference is that Clock implicitly updates values through requests. Programmers must explicitly send updates up the tree (towards the application). Clock then sends request values back down the tree (towards the user), after evaluating constraints.

For any Clock implementation we can derive a LOTOS interactor network. We can specify the behaviour of these interactors in equivalent LOTOS. We must translate functional data types into Act One. The development of E-LOTOS [Jeffrey1996] will make this translation easier. We are currently developing tool support to make this translation automatic.

4.5 Proof

Once we have a LOTOS specification we can attempt to verify properties about our specification. This can help us to guarantee critical properties about our design. However, to be pragmatic we must focus this verification effort. In this section, we show how we used proof to guarantee important properties about our system design that were shown to be important in our earlier UAN task analysis (see sections 4.2.2, 4.2.3).

We can perform this verification in one of two ways, using LOTOS simulation or model checking technology. Markopoulos [1997] has shown how the LOLA simulation tool can be used to demonstrate task conformance between a LOTOS task specification and system specification. We have made use of similar tests to demonstrate task conformance with our implementation.

We have also made use of model checking technology to verify properties about our design. Model checking provides a powerful approach to formal verification. Given some specification (that can be translated into a Finite State Automata), and some temporal logic formula, a model checker can perform a fully automated proof of whether the specification satisfies the formula. This makes it easy to perform complex proofs in the context of iterative design. If the specification is altered, performing the proof again requires only that the temporal formula be checked automatically against the new specification. This form of rechecking can be as automatic as using regression tests in software design. This is in contrast to Theorem Proving, where a whole series of lemmas may be required, before a formula can be completely proved. However, model-checking environments also tend to have some disadvantages. In general, they can only perform proofs across finite systems, using finite types. We therefore need to transform our LOTOS specification into a finite instance. For instance, with our QOC editor specification we prove properties about an instance where there are only a finite, and given number, of users, nodes and edges in existence. Tool support can again help here in producing such a finite specification.

We have made use of the Caesar/Aldebaran toolkit (CADP) [Garavel1996] which makes use of the mu calculus [Garavel1996] to perform verifications. For instance, with it we can verify that our system implementation supports the locking properties that were highlighted in our task analysis:

- **No two users can alter the same node at the same time;**
- **If one user has locked a node it cannot be deleted by another user, even with delete group.**

The first statement can be expressed in temporal logic as follows:

ALL (

["SelectNode !1 !1"]

SU (["SelectNode !1 !2"]F) (<"ReleaseNode !1">T)

)

This statement says that , in all states, if user one selects a node no other user can select that node, until user 1 releases that node. The statement ["X"]F means that in the current state it is impossible to perform "X"; the statement <"X">T means that in the current state it is possible to perform an "X". The mu calculus allows us to express proofs over events with specific data. It does not, however, allow us to express statements over an event with any data. We must therefore verify our specification over a restricted, but sufficient set of users and nodes.

We can do something similar for our second requirement:

ALL (

["SelectNode !1 !1"]

SU (["DeleteGroup !(SET 2) !2"]F)

(<"ReleaseNode !1">T)

)

This statement says that, in all states, if one user has selected a node, then another user cannot delete a group containing that node, until the node has been released.

Though this formal verification still cannot guarantee that our design is perfect, it can enable us to have more faith in the correctness of our design. The use of tool support in deriving a LOTOS specification and performing model checking makes this activity easier and more cost effective. The strong link between the task analysis and system proof is also important. We have not tried to perform exhaustive proofs about our design, instead we have focussed upon usability requirements highlighted in the earlier task analysis.

6 Conclusion

In conclusion, the development of interactive systems requires a variety of modelling approaches. The DSVIS'97 workshop proceedings stressed this point, suggesting that some of the more formal methods described in previous DSVIS proceedings, are unsuitable for early stages of a design and analysis [Fields1997b]. Instead we require a variety of lightweight models that can be used to give multiple views on a system [Fields1997a]. In this paper, we have shown how some lightweight models can be used in a design. To do this properly, we need tool support to link these multiple views together, to enable designers to see how they relate to each other and the design at large [Clark1997]. Brown et al [1998] have taken a step towards solving this problem by providing tool support to help link UAN specifications to Clock prototypes. This makes it easier to see where changes in one model must result in changes in another.

High-level prototyping languages, such as Clock, can help in this development process. They provide a means to produce rapid and well structured prototypes. The use of constraint based programming allows a level of abstraction closer to that provided by state based modelling approaches [Harrison1994], but also allows immediate user testing.

The ability to derive LOTOS specifications, from a Clock prototype, and then reason about the specification, all in a tool supported manner makes the use of formal specifications more cost effective. The development of a full LOTOS specification requires considerable time and knowledge; tool support can help to simplify this process.

Now that we have demonstrated the initial feasibility of our approach, there are some remaining issues that need to be addressed.

- Coverage - we have a good but incomplete coverage of the design life cycle. We now intend to look at how other modelling approaches could fit in with this approach, particular user modelling which may help in performing error analysis on an initial design.
- Training & Cost - we need to demonstrate that our method is really usable by others. We have made use of a variety of pre-existing techniques. This may help here. However, the range of techniques used may still make the approach too costly in terms of time and training.
- Iterations - we need further consideration about how the different design stages link together. We also intend to perform user testing upon the artefact we have produced to investigate how usability evaluations fit into the design cycle.
- Team work - since design is a group activity, we need to properly investigate how easy it is for groups of designers to work together using our approach.

References

G Abowd and A Dix (1992), Giving undo attention. Interacting with Computers, 4(3):317-342.

Heather Alexander (1990), Structuring dialogues using CSP, in MD Harrison and H Thimbleby, *Formal methods in Human computer interaction*, Cambridge University Press.

R Bentley (1994) Supporting multi-user interface development for cooperative systems, PhD thesis, Lancaster University.

I.M. Breedvelt-Schouten, F Paterno, C Severijns (1997) Reusable structures in task models, in proceedings of DSVIS’97, pp 225-241.

Judy Brown, T.C. Nicholas Graham and Timothy Wright (1998). The Vista Environment for the Coevolutionary Design of User Interfaces. In Proceedings of Human Factors in Computing Systems (CHI'98), ACM Press, Los Angeles, USA, April 1998 (to appear).

Steven Clark (1997), Literate Development, PhD thesis, University of Glasgow.

Conklin J and Begeman, ML (1989) gIBIS a tool for all reasons, in J. Amer. Soc. Info. Sci 200-213.

Alan Dix and Gregory Abowd, (1992) Integrating status and event in formal models for interactive systems, in Software Engineering Journal, 11(6) pp 334   347.

A. Jeffrey and G.Leduc (1996), E-LOTOS core language. Output of the Kansas City meeting, version 1996/09/20, (ISO-IEC/JTC1/SC21/WG7).

Fields R, Merriam N, Dearden A (1997a) DMVIS: Design, modelling and validation of interactive systems, in Proceedings of DSVIS'97, pp 29-45.

Fields R, Merriam N (1997b) Modelling in Action. Reports from the DSVIS'97 working groups, in Proceedings of DSVIS'97, pp307-320

Hubert Garavel (1996), An Overview of the Eucalyptus Toolbox. In Z Brezocnik and T Kapus, eds, Proceedings of the COST 247 Interanational Workshop on Applied Formal Methods in System Desing (Maribar, Slovenia), pages 76-88. University of Maribor, Slovenia, June 1996.

T.C. Nicholas Graham and Tore Urnes (1996a). Linguistic Support for the Evolutionary Design of Software Architectures. In *Proceedings of the Eighteenth International Conference on Software Engineering*. IEEE Computer Society Press, Berlin, Germany, pp. 418-427, March 1996.

T.C. Nicholas Graham, Herbert Damker, Catherine A. Morton, Eric Telford and Tore Urnes (1996b). The Clock Methodology: Bridging the Gap Between User Interface Design and Implementation. York University Technical Report CS-96-04. York University, August 1996.

Phil Gray, David England and Steve McGowan (1994), XUAN: Enhancing the UAN to capture Temporal Relations among Actions, Technical Report IS-94-02, Department of Computing Science, University of Glasgow.

Thomas R Green (1989) Cognitive dimensions of notations. In A Sutcliffe and L Macaulay, eds, People and Computers IV, pages 443-460. Cambridge University Press, Cambridge, United Kingdom.

Michael D. Harrison and David J. Duke (1994), A review of formalisms for describing interactive behaviour, Amodeus Project Document: SM/WP28.

H.R. Hartson, A.C. Siochi and D Hix (1990), The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. ACM Transactions on Information Systems, 8 (3) :pp191-203.

Ralph Hill. (1992) The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In ACM SIGCHI 1992, pages 335-342, April 1992.

CW Johnson (1996). Literate specifications in Software Engineering Journal, July 1996, pp 225-237

CW Johnson (1997) Utility of User interface notations, submitted to the *Journal of Human Computer Interaction*, 1997.

P Johnson, S Wilson, P Markopoulos & J Pycock (1993) - ADEPT - Advanced design environment for prototyping with task models, Demonstration abstract. In Aschlund S et al (eds), Bridges Between Worlds- INTERCHI'93, Addison-Wesley, pp 56.

Glen E Krasner and Stephen T Pope (1988), A cookbook for using the Model-View-Controller interface paradigm. *Journal of Object-Oriented Programming*, 1 (3):26-49.

A Maclean, R Young, V Bellotti, and T Moran (1996), Questions, Options and Criteria: Elements of Design Space Analysis, in Moran (1996).

Panos Markopoulos (1997), A compositional model for the formal specification of user interface software, PhD Thesis, QMW College, University of London.

TP Moran , JM Carroll (1996) (eds) Design Rationale: Concepts, techniques and use, Hillsdale, Lawrence Erlbaum Asoociates.

Brad A Myers, Dario A Giuse, Roger B Dannenberg, Brad Vander Zanden, David S Kosbie, Edward Pervin, Andrew Mickish and Philippe Marchal (1990), Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. In *IEEE Computing*, pages 71-85, Novermber 1990.

F Paterno and M Mezzanotte (1995), Formal verification of undesired behaviours in the CERD case study, in proceedings of EHCI'95, Chapman & Hall Publisher.

Pavon S & Larrabeiti D (1993) LOLA (LOtos LAboratory) User Manual v3.4, http://www.dcs.upm.es/~lotos

M Sage and CW Johnson (1997) Interactors and Haggis: Executable specifications for interactive systems, in proceedings of DSVIS'97, pp 93-109.

S Shum (1991) Cognitive dimensions of design rationale. In D Diaper and N Hammond, eds, People and Computers VI: Proceedings of HCI'91. Cambridge University Press, Cambridge, United Kingdom.

Shum, S (1993) QOC Design Rationale Retrieval: A Cognitive Task Analysis & Design Implications, Rank Xerox EuroPARC, Technical Report EPC-93-105

S Buckingham Shum (1996) Analyzing the Usability of a Design Rationale Notation, in Moran (1996).