Specifying Collaborative Software: A Proposal

Steve Reeves
Department of Computer Science
University of Waikato
Hamilton
NEW ZEALAND
stever@waikato.ac.nz

Abstract

The aim of this paper is to illustrate how formal specifications for collaborative interactive systems might be written. It presents a new modelling paradigm for certain systems. It also shows how formal software engineering approaches can be useful. Specifically we choose to specify a simple collaborative editor. This example serves two purposes: it shows how clear and simple a formal specification can be and it provides a basis for making observations about the requirements for a specification language where the target is CSCW systems. The specification of the system has three parts: the semantics of the system; the syntax of the system; the semantics of the collaborative aspects of the system.

1 Introduction

The aim of this paper is to recount our exploration for a way of writing formal specifications for collaborative interactive systems-one outcome of this exploration has been a three-part framework which serves to characterize such systems and which we propose as a useful model.

Computer supported collaborative work (CSCW) has all of the complications, subtleties and problems of any interactive system plus the added interest of having to deal with the fact that people are collaborating. All this gives us the opportunity to cover many conventional problems with specifying systems but also allows us to suggest some extensions.

In order to illustrate our proposal we use as an example a simple collaborative editor. As should be clear, the languages we use and the framework within which we work can be used on many different problems with far greater complexity than the one we have chosen as a vehicle in this paper. However, the editor is complicated enough to allow us to show all the aspects of the framework and languages we are proposing while at the same time not being so complicated as to obscure the more general points we are making.

The work described here is, as will quickly become apparent, closely based on the work of Hussey and Carrington in [3]. To set the scene (and to allow the current paper to be reasonably self-contained) we need to reiterate some of the points made in the introduction to [3].

A specification describes what a system provides for its users but has nothing to say about how these provisions are implemented, which is the task of the design and implementation phases of development. This is because the job of a specification is to make as clear as possible what a system does, which requires the freedom to pitch the description at a level of abstraction which most closely fits a natural way of thinking about the system. It avoids forcing our thinking too early into the straight-jacket of a language developed for the instruction of a machine. In particular, as Hussey and Carrington say, "a user-interface specification does not describe the perceptual (visual/audio) aspects of user interfaces because such a description represents design." We are interested in formal specification for many reasons, all of which have been extensively debated. We will not rehearse those debates here: suffice to say that the reasons include precision and lack of ambiguity, conciseness, a basis for contractual agreements between client and provider, a basis for building confidence in correctness and robustness (especially important for safety- and security-critical systems), a basis for various tools for supporting the development process (animators, proof assistants, editors, language checkers) and, finally, as basis for correct and complete document and on-line help.

In the next section we introduce the example system that we will use as a running example throughout the paper which, though simple, does exhibit enough of the typical attributes of a CSCW system to allow us to show the need for the methods we are proposing. Since

O-8186-8828-9/98 $10.00 © 1998 IEEE

the language we will be using is an adaptation of Z itself called Object-Z (see [6]), which is not very commonly known, we will also spend some time explaining the specification in detail.

The specification of a system, as we propose it, has three parts: the semantics of the system; the syntax of the system; the semantics of the collaborative aspects of the system.

The first two of these follow Foley and van Dam's own three-part model of an interface implementation as adapted by Hussey and Carrington. The semantic aspect of a specification associates an abstract meaning with syntax; the syntactic aspect defines valid sequences of user-interaction with the system. The third aspect (which is not part of specification) is lexical: it describes the mouse movements and clicks, keyboard presses and how the system is presented to the user. Defining the lexical aspect of a system is a design activity since it involves deciding on and describing how the system looks to the user. This section and section three consider the semantic and syntactic aspects of our example system.

Our third aspect of the specification is required in order to say what awareness users should have of, or how they should be informed about, each other and the system. This is a new requirement that comes from the fact that we are dealing with collaborative systems in which users need to be aware of one another. In particular they need to be aware of each other via the medium of the system and it is this mutual awareness via the system that the third part of the specification has to describe. This is considered in section four.

In section five we consider how the methods described could be made fully formal.

The view that we take of work in software engineering in general (which should include HCI) and on user-interfaces in particular follows very closely that propounded in [7], especially where the author says "By means of formality, and abstraction in particular, we can focus on and reason about the subtleties and details of using computers. Without formality it would be far too easy to get lost in the intricacies of particular implementations." (page vi).

2 A Shared Text-editor

Imagine that our task is to write a formal specification for a shared-text editor. We choose to present the solution to this task in the conventional way for Z-style specifications by giving the formal parts of the specification followed by informal descriptions.

In order not to make the problem harder than it needs to be to make our points (and in order to fit into the space available) we have to make several simplifying assumptions, as will become apparent.

First, there is a basic type that everything is built on

$$[Char]$$

which introduces the set ***Char*** which contains all the characters that a user will need when writing their documents. There is another basic type which allows us to talk about the users:

$$[People]$$

introduces the set ***People,*** which is no further defined since we do not need to know about any properties of people for this specification.

The editor itself is a system which people (from ***People)*** can join as users. Once they are users they can participate in the editing of the shared document. All users are aware at any time of the contents of the document; in other words there are no restraints on merely viewing the document. However, to insert or delete text, or to even place a cursor in the text, a user needs to lock a paragraph (which we take as the atom out of which a document is composed: clearly alternative decisions could have been made about this) to protect it from similar, simultaneous use by other users. The text is a sequence of characters of type ***Char.***

Having introduced the basic sets and given an informal description of the editor we can now turn to the main part of the formal specification.

The idea of a paragraph of text is captured in a class which contains all the expected state and functions for modelling a paragraph of text, as shown in Figure 1. (This part of our example is based on the text editor given in chapter 18 of Diller [2]).

The state of an object in the class ***Paragraph*** is given by two sequences of characters, one representing all the text to the left of the current cursor position and the other representing everything to the right.

When an object in this class is created the initialization operation $INIT$ is performed, which in this case makes sure that the paragraph is empty of text since $\langle\,\rangle$ denotes the empty sequence. Note that each paragraph has a cursor and it is this feature that allows several users to edit the document at once. This class has six operations defined in it, each defined by a schema. Taking the schema for $InsertLeft$ as an example: $\Delta(left)$ indicates that the only state variable changed by the operation is left; ***char? : Char*** declares the local (to this operation) variable ***char?*** to be of type ***Char;*** the ? indicates, by convention, that this is an input variable; below the dividing line is the predicate part containing

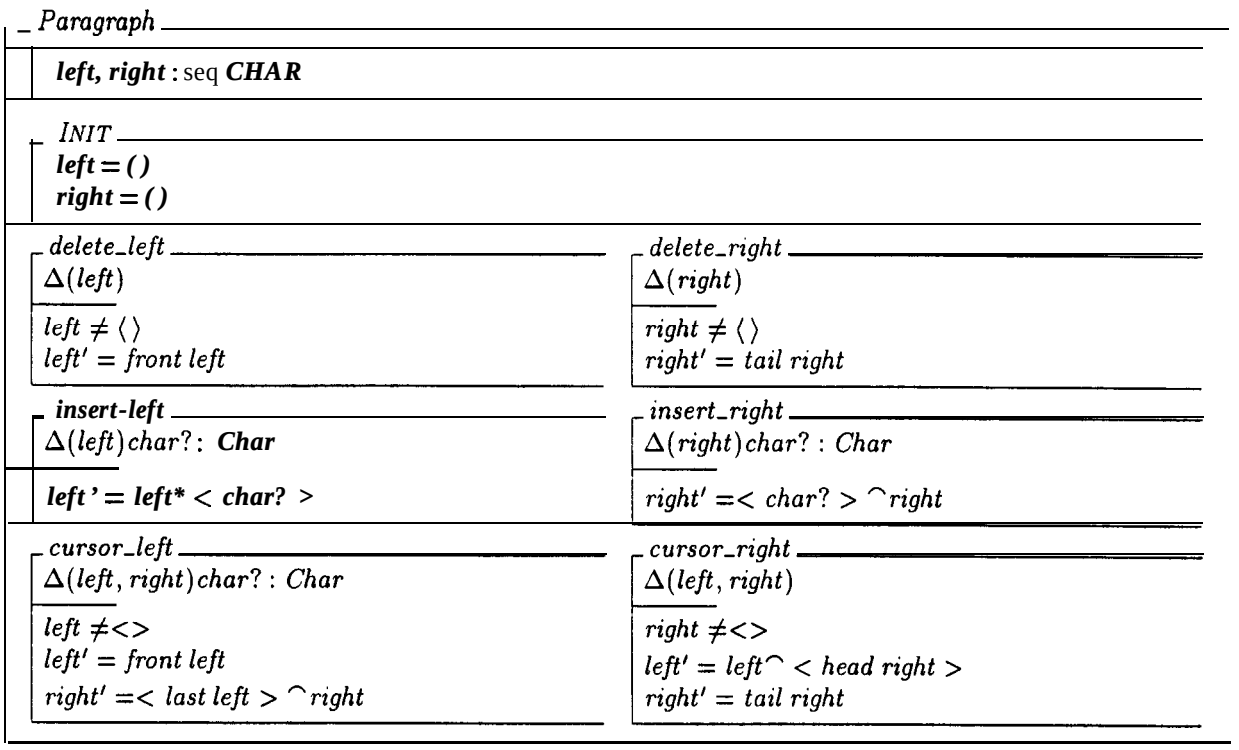


Figure 1: The ***Paragmph class***

logical statements which have to be true of this operation, so in this case ***left' = left ⌢ (char?)*** indicates that the value of the state variable ***left*** after the operation (denoted by the fact that the variable is primed) is equal to the value of ***left*** before the operation with ***char?*** appended to its right-hand end. Recall that ***left*** is a sequence of characters.

Having specified the class of paragraphs we have to look at how these are assembled to make a document that is to be edited and also how the fact that there may be several people editing the document at once can be modelled. In fact, an editor will be modelled by putting together both these components.

We choose to view a document simply as a sequence of objects from the class ***Pamgmph. So,*** each paragraph of the document being edited will have within it the text being edited and the current cursor position (modelled by the object's state) and the person editing that paragraph will be able to perform the operations associated with the object.

The editor needs to model not only the document being edited but also the interaction that each user is making with the document. In this example we choose to do this by allowing each user to lock a paragraph, as long as no other user has locked it, and restricting users to doing operations involving the cursor to paragraphs they have locked.

All this is modelled by an object in the class ***Editor,*** shown in Figure 2.

Note that the fact that a user may have several paragraphs locked at once is specified very precisely and concisely by the type of ***locked-by.*** Since the type given is of a partial function with no further constraints it means that not every paragraph has some locking user associated with it (partiality) and that each paragraph has at most one locking user (functionality). The only paragraphs that can be locked are those that appear in the document ($\mathrm{dom}\ locked\_by \subseteq$ ranparas) and the only people who can lock paragraphs in the document are users (ran ***locked-by*** $\subseteq$ ***users).*** The initial state of the editor is one where there are no locked paragraphs.

A paragraph can be taken by a user only if it is not already locked ($para? \in$ (ran $paras$) $-$ (dom ***locked-by))*** and then the ***locked-by*** function is updated so that it now records the paragraph as being locked by the user ***(locked-by = locked-by*** $\cup$ ***pam?*** $\mapsto$ ***user?).*** A paragraph can only be dropped by the user who locked it ($para? \mapsto$ ***user?*** $\in$ ***lockedby).***

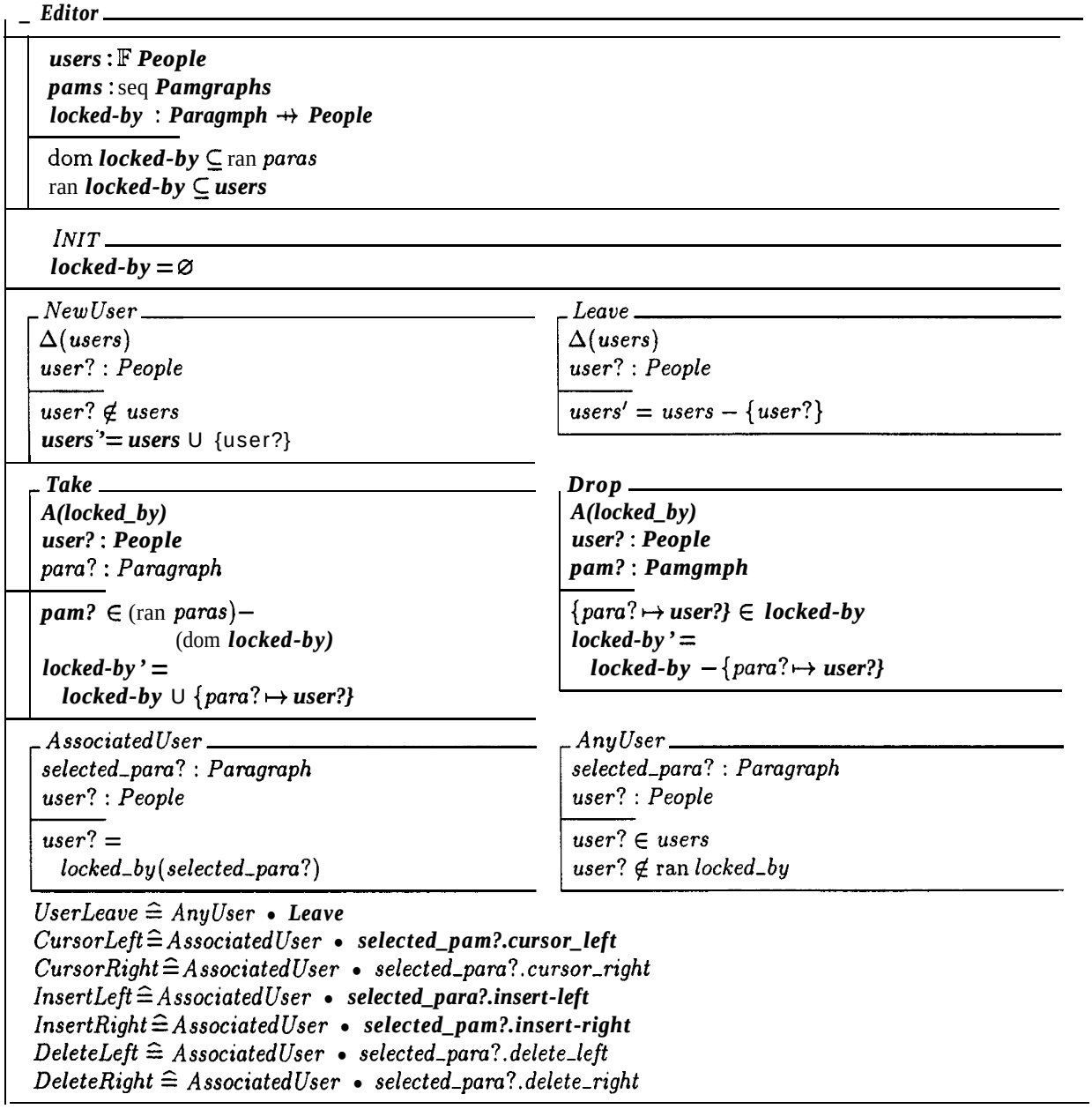


Figure 2: The ***Editor*** class

3 Dialogue

The above specification defines the semantics of the system. It says what the meaning of the operations performable by the system are in terms of their effect on the state.

The next thing we have to do is to specify what interactions with the system are possible in terms of what sequences of operations can be requested by the user when they are using the system. That is, we now have to deal with the syntax of the system. As in [3] we use CSP definitions to specify what sequences of operations count as allowable interactions with the system. First the alphabet of the editor, that is the set of allowable requests for operations to be performed, is defined as

$$\alpha\ EDITOR = \{New\ User, UserLeave, Take, Drop, CursorLeft, CursorRight, DeleteLeft, DeleteRight, InsertLeft, InsertRight\}$$

Note that the operations which are part of objects from the class *Paragraph* are not allowed operations as far as a user is concerned.

Next, we have to specify what sequences of operations the user can properly request of the system. We do this by defining a process *EDITOR* which specifies the interactions allowed at the outer level, that is with the editor as a whole.

$$EDITOR = NewUser \rightarrow EDIT$$

This means that the only operation a user can request when they first use the editor is *NewUser*. Having successfully completed that operation, the system then behaves as the process *EDIT:*

$$EDIT = Take \rightarrow EDITPARA \mid Leave$$

Here the user has a choice: they can request a *Take* operation and if the operation completes successfully the system will behaves as the process *EDITPARA;* or they can request the *Leave* operation and if that completes successfully there are no further editor operations that they can carry out.

$$EDITPARA = x : \{CursorLeft, CursorRight\} \rightarrow EDITPARA \mid EDITTEXT \mid Drop \rightarrow EDIT$$

Here there is a choice: either the user can do one of *CursorLeft* or *CursorRight* or they can proceed directly with the process *EDITTEXT* or they can do *Drop* and the system behaves as the process *EDIT*.

$$EDITTEXT = x : \{DeleteLeft, DeleteRight, InsertLeft, InsertRight\} \rightarrow EDITPARA$$

Here the user can do one of the text-changing operations and then proceed with the process *EDITPARA*. *So,* the syntactic specification explicitly defines what sequences of interactions may be possible. For example, note that a user cannot alter the state of a paragraph until they have locked it. In fact, this could have been inferred from the semantics of the system since the operation *InsertLeft*, for example,

$$InsertLeft \mathrel{\widehat{=}} AssociatedUser \bullet selected\_para?.\ insert\text{-}left$$

requires that the operation *AssociatedUser* is carried out and that is only possible if the user has locked the selected paragraph, which is enforced by the predicate part of *AssociatedUser*. Equally, the explicit fact that *UserLeave* is available to any user who has no paragraphs locked is implicit in the definition

$$UserLeave \mathrel{\widehat{=}} AnyUser \bullet Leave$$

However, there is certainly no harm in this redundancy. Indeed, it is likely that we can make a definite virtue of it since the interactions are made explicit in the syntax and so are more easily seen to have been provided by the system. The fact that they have been correctly reflected in the (perhaps more complicated and more subtle) semantics of the system can then be checked and proved to hold, given suitable proof support.

Note that the fact we are specifying means, as ever, that we want to say *what a* user can do by way of interacting with the system, but not *how* they will do it. As long as the designer, when deciding how a user shall interact with a system-for example with a mouse and menus, via typed text on a command line, using a toolbar etc.-satisfies the constraints put in place by the specifier, they are free to use whatever mechanisms they feel will be most useful or suitable.

4 Global Collaborative Properties

The specification above certainly describes the functions of the system and the allowable interactions. However, there is still something missing: we have not

yet said anything about how the fact that this is a collaborative system, possibly with several users, should affect the specification. For example, we have not said what other users should be informed of when a new user joins the collaboration. Neither have we said what other users should be informed of when a user locks a paragraph, for example.

Many of these sorts of conditions on collaboration could be coded into the semantics in much that same way as many of the constraints on allowable interaction were. However, for the same reasons that we prefer to explicitly specify the allowable interactions using the methods of the previous section, we also prefer to specify the collaborative aspects of the system explicitly. Again, this promotes clarity, may introduce some redundancy (so providing checks that things that are required have actually been specified) and also gives us two formal statements whose correctness relative to one another we might wish to prove. All of these facts serve to encourage specifiers to think about the system they are describing and make checks that everything is as it should be.

The language we use to specify these global properties of the system is an agent-action logic as described in [4]. For example, assuming that we have a predicate ***present*** where ***present(x)*** is true if and only if person x : ***People*** is present as a user of the editor (which is another way of saying x $\in$ ***users*** within an object in the class Editor) then we could write

$$\forall x : \textbf{People} \bullet \textbf{(present(x)} \Leftrightarrow \text{V y} : \textbf{People} \bullet \textit{informed}(y, \textbf{x))}$$

which specifies that some user is present if and only if all other users are informed of their presence. Also, the fact that all users should be informed of the contents of the document can be specified by

$$\text{Vx} : \textbf{People} \bullet \forall p : \textbf{Paragraph} \bullet \textbf{present(x)} \Leftrightarrow \textit{informed}(x, p.left \frown \textbf{p.right)}$$

In general we will want to write statements of the form

$$S\,(P \Leftrightarrow [a,\ \alpha]Q)$$

where S is some signature or declaration of variables and their types, ***P*** is some precondition, ***a*** is some operation, a is some agent which performs the operation ***a*** and Q is some postcondition. The intended meaning is that for each of the variables in S, if ***P*** is true then whenever ***a*** is successfully completed by α then Q will be true. Here is an example of this form: if a user locks a paragraph we probably want all users to be informed of that fact and vice ***versa:***

$$\text{V x, } y : \textbf{People} \bullet \forall p : \textbf{Paragraph} \bullet \textbf{present(x)} \Leftrightarrow [\textit{Take}(x, p), x]\textit{informed}(y, \textbf{lockedby(p) = x)}$$

This means that if x is some user then whenever the operation ***Take*** is successfully carried out by them on some paragraph ***p*** then every user will be informed that ***p*** is locked by x.

In order to make these sentences easier to read we introduce some two-dimensional structure to them (like the structure we see in Z and Object-Z). The example above becomes

x, y : ***People*** ***p*** : ***Paragraph***	signature
present(x)	precondition
Take(x,p) | x	action | agent
$informed(y,$ ***locked-by(p) = x)***	postcondition

We see that again we are able to write very abstract specifications which are also completely precise and unambiguous. Here is another example:

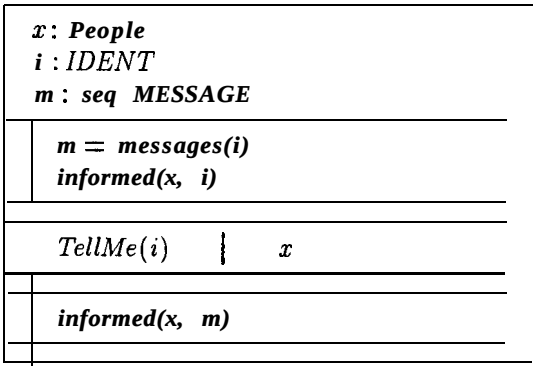
x, y : ***People*** ***p*** : ***Paragraph***	
present(x)	
$Drop(x, p)$ I x	
$informed(y, \neg$ ***locked-by(p) = x)***	

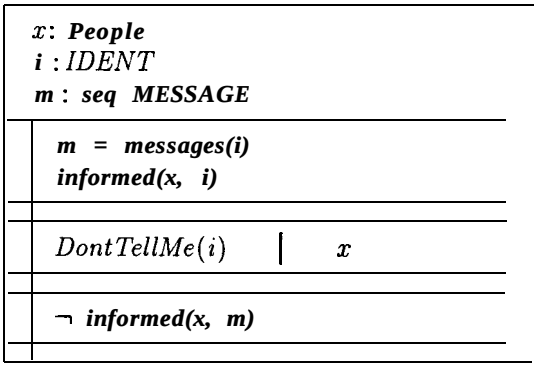
which specifies that whenever some user drops a paragraph everyone is informed that the paragraph is no longer locked by them.

We can use this language to explore the idea of specifying policy issues too. For example, assume that there is some part of the system which records messages from users on different subjects. We model this by

$$[IDENT,\ \textbf{MESSAGE}]$$
$$\textbf{messages} : IDENT \rightarrowtail \text{seq } \textbf{MESSAGE}$$

where $IDENT$ contains the names of subjects about which ***messages*** (from the set ***MESSAGE)*** are sent and stored. Now we can specify operations $TellMe$ and $DontTellMe$ thus:





Much more can be said about this style of specifying operations that affect informedness, but we leave that for another time since we now need to consider some more general concerns.

5 Formal concerns

Though we can give a well-defined semantics to almost all of the above, there is still a gap which makes it informal and descriptive, i.e. we could not, for example, do any proofs relating the sentences above to each other. The reason for this is that we have not given a formal definition of the predicate ***informed.*** This is not to say, of course, that the framework we have proposed and the languages we have used to illustrate the examples above have no value-given that there are no other frameworks currently proposed which have even the level of formality we have made available (which given the fact that the three components of our framework are, as languages, independent means that should we want to we can do formal reasoning in the syntactic and semantic phases and also within the collaborative phase in the absence of ***informed) we*** have at least made progress there-but the fact that we have given precisely defined languages means that we can immediately benefit from the usual positive points within formal methods of precision, conciseness and the possibility of a basis for prototyping by providing interpreters for the languages used above.

Even given the absence of a semantics for ***informed,*** if we intend to use this language as part of the specification of a system then this is not likely to cause problems. The designer will simply have to decide whether their proposed step towards implementation does actually fulfill some informal notion of having informed a user of the value concerned-we shall at least been able to communicate this part of the specification precisely and unambiguously.

Examples of how a given designer or implementor may choose to define ***informed*** are:

informed(x, y), where x and y are users, is made true by having a scrollbar for each user y appearing on the desktop of user x;

informed(x, y), where x and y are users, is made true by having a window appearing on the desktop of user x showing explicitly (perhaps by the presence of a name or a cartoon or photograph or a real-time video image of them) that user y is using the editor;

informed(x, p), where x is a user and ***p*** some information to be made available, is made true by having a window appear on the desktop of user x which represents the information p-perhaps by displaying text if ***p*** is textual or a diagram or a photograph if ***p*** is diagrammatic or pictorial.

We have to remember that the essence of the job of the specifier is not to say which of these should be used but that one of these (or some other means) should be used to satisfy ***informed.*** The decision as to how ***informed*** is satisfied is decided on by the designer since that is part of the activity that takes place during the design, and the designer must not have these design decisions pre-empted by the specifier. As ever we have to keep in mind that specification says ***what*** should hold (i.e. that ***informed*** is true) and it is not until we get to design that ***how*** something is made to hold is decided on.

As we said above, if we want the language to be formal, so that we can do proofs involving it, for example, then we need to go further. One way we could make progress is to develop some axioms for informedness, or perhaps more generally 'awareness', so that the language has a formal meaning.

We might start from work on ideas like knowledge which are, probably, close to the idea of awareness. However, trying to formalize knowledge has kindled a whole hard, interesting and very active area of artificial intelligence where the debates are still raging. A second approach is to argue that while awareness has some things in common with knowledge, it is a

much less complex idea and perhaps starting from scratch and trying to assemble (guided by naive work on knowledge) might be successful. In work on logics of knowledge the logic S4 is sometimes held up as a starting point since its axioms are few and simple and (attractively) appear to embody some aspects of knowledge. One example axiom is $knows(x, a) \Rightarrow knows(x, knows(x, a))$ where x and a range over agents and predicates respectively. It is not hard to see objections to this axiom in the case of knowledge since it states that everyone knows what it is that they know, i.e. that we are all, in some sense, perfectly introspective. However, if we think about the awareness version of it $aware(x, a) \Rightarrow aware(x, aware(x, a))$ then it is arguable that this does capture some of the sense of awareness. It is hard to see how you could not be aware of something that you are aware of. One aspect of the difference between knowledge and awareness is that awareness is surely a conscious state while much knowledge gets deeply buried.

This is not the place to take these ideas any further-that is done in [5], however, where the investigation of possible logical frameworks is considered in some detail. However, the conclusion that seems to be emerging from this work is that the idea of awareness has so few logical properties that a search for an axiomatization (or other formalisation) of it is doomed to failure.

6 Conclusions

We have introduced and described a three-part model for specifying CSCW systems. The three parts specify: the allowable interactions a user may have with the system, the system's syntax; the meaning of the components of these allowable interactions, the system's semantics; what the user should be made aware of when they are using the system. The system we have described here is quite simple, yet we have been able to introduce and discuss many aspects of the specification of a CSCW system.

We have, of course, taken a very naive view of a collaborative editor. Work is currently in progress to apply to the method outlined above to realistic designs, drawing in particular on some of the excellent work by such, amongst others, as Beaudouin-Lafon and Karsenty [1] on usability aspects of collaborative editors.

Once a system has been specified we need to use the formality of the specification to support activities like: making sure (perhaps by proof) that the system has the properties required of it; that operations which are intended to fit together to form larger operations really do fit properly (for example, are the assumed outputs of one operation really what the next operation is expecting as input); using the formal specification as a basis for complete and clear documentation; taking the opportunity of having to describe the system precisely (just as we do when coding it) to think it out properly (but at an abstract level, so that the system is more clearly seen).

A major point of this sort of work is to describe precisely what the system does without any unnecessary constraints on the designer when they start to decide how the system should do things. This can be a hard distinction to grasp and it can be even harder to reflect in our actions. However, if we are to have any hope of precisely describing what complex systems do, we must seek to retain the distinction.

Acknowledgements

I am grateful to Hugh Anderson, Lindsay Groves and Ken Robinson for recent conversations which has made the need to look more pragmatically at the idea of awareness all the more clear and for their suggestions that pursuing the more pragmatic ideas around informedness are likely to be more fruitful.

References

[1] M. Beaudouin-Lafon and A. Karsenty. Transparency and awareness in a real-time groupware system. In Proceedings of *UIST'92*. ACM Press, 1992.

[2] A. Diller. ***Z:An Introduction to Formal Methods (2nd. ed.). J.*** Wiley and Sons, 1994.

[3] A. Hussey and D. Carrington. Using Object-Z to specify *a* web browser interface. In ***Proceedings of*** *OzCCHI'96.* IEEE Computer Press, 1996.

[4] S. Reeves. Specifying and reasoning about CSCW. In F. Bodart and J. Vanderdonckt, editors, ***Proceedings of 3rd. Eurographics Workshop*** *DSV-IS'96*, pages 366-383, Berlin, 1996. Eurographics, Springer-Verlag.

[5] S. Reeves. Formalizing awareness: A survey of some possibilities. In M.D. Harrison and J.C. Torres, editors, ***Proceedings of 4th. Eurographics Workshop*** *DSV-IS'97*, pages 341-358. Eurographics, 1997.

[6] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, ***Object Orientation in Z,*** Workshops in Computing, pages 59-77. Springer-Verlag, 1992.

[7] H. Thimbleby. User ***Interface Design.*** ACM Press and Addison-Wesley, 1990.