ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

A Modal Model of Versions

A. Dix, T. Rodden and I. Sommerville

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76105-5

A Modal Model of Versions

Alan Dix

School of Computing, Staffordshire University,
Stafford, ST18 0DG, UK. email: A.J.Dix@soc.staffs.ac.uk
url: <http://www.soc.staffs.ac.uk/~cmtajd/>

Tom Rodden and Ian Sommerville

Collaborative Systems Engineering Group, Department of Computing
Lancaster University, Lancaster, LA1 4YR. E-mail: {tam,is}@comp.lancs.ac.uk
<http://www.comp.lancs.ac.uk/computing/staff/{tam.html,is.html}>

Abstract

This paper is concerned with the problem of version management for cooperative systems. We present a model of versions which is a 'modal' model in that it takes into account the context of use of these versions. We describe the basic version model, the modelling of relationships and the modelling of version management.

1. Introduction

Most prototype CSCW systems are based on a view of use which is oriented towards the experience and the work setting of their developers. In essence, existing cooperative applications seem to assume that they will be used in work settings comparable to research labs where the system users have a great deal of individual autonomy and have established working practices which largely depend on a shared understanding of the work which people do. However, this is quite different from many real commercial settings where the organisation demands control over work products. The lack of recognition of control requirements has, we believe, limited the growth of groupware products.

Limited effort has been applied to establish an understanding of the relationship between flexible and open cooperative systems and the managed commercial domains within which they are to be placed. An area where this mismatch is particularly evident is in the lack of support for the evolution, management and co-ordination of the products of collaborative endeavours. We are not aware of a generic model of version management for cooperative applications. However, in many organisations, it is imperative that procedures should be established to manage just this activity. We are currently exploring the requirements for version management in cooperative systems; in this paper we present some results of this research namely a generic version model suitable for application within cooperative settings.

Although superficially straightforward, version management has some surprising subtleties. Typically, a version manager will capture some, but not all versions of a particular entity. For example, you might annotate a printed version of a document and then type in some, but not all the annotations. The annotated copy is clearly a version of the document, but not one that is ever captured electronically. It is therefore never part of an electronic version manager. In general, a particular version manager will only capture some part of the full versioning picture. Although the model we construct is not limited to what a version manager *can* capture, it must be capable of modelling what version managers *do* capture. It is not just model of versions, but of version managers (which are themselves a partial model of versions!).

This paper reconsiders the notion of versions and introduces a model of versions that make explicit the context within which the version is exploited. That is, we model what versions are in the real world. The proposed modal model can then be used to model the activity of version managers.

2. Background

The core of our consideration of versions is the problem of version management and its role in configuration management. Most existing work on configuration management focuses on the control of large software projects. To support the configuration management process, many different software tools have been produced. Tools such as SCCS [1], and RCS [2] support version management by optimising the storage requirements of multiple versions. They provide check-in and check-out facilities which ensure that only a single copy of a controlled item may be modified by one person at one time.

Both SCCS and RCS are limited by a very simple model of version identification where versioned entities are allocated numeric identifiers. When a version is to be retrieved, this record must be consulted to find the version name. This is an error-prone process once there are more than a few versions. To counter this

problem, version management systems have been developed which support attribute-based version identification. Therefore each version is identified by a set of attributes such as *author*, *creation-date*, *language used*, etc. Some of these attribute-based systems can link versions with the software process activities responsible for their creation [3,4,5]. Rather than forcing the retrieval of a specific version using some arbitrary name, these systems allow the characteristics of the version to be used to identify it.

Most version management is prescriptive with an onus on controlling and centralising the version management process. There has been little research into the problems of versioning in shared settings where the demands of more than one user need to be addressed. Work that has occurred to date focuses on the merging of versions produced by different members of a group [6]. This has been concerned with merging independently made changes to software components. The version models adopted by most existing version management systems have also sought to preserve the illusion of a single-user virtual machine. This illusion is, however, problematic to most approaches supporting collaborative work.

It may at first sight seem attractive to maintain an individualistic model from a control point of view. It means that many parallel group activities need to be serialised when mediated by such systems. However, this may force users to work outwith the system causing the underlying version model to be compromised. Where versions have been important in CSCW systems application developers have been forced to develop their own model of versions. For example, a version model has been developed as part of a hypertext co-authoring system [7]. The developed model is closely bound with the supporting application. It focuses on supporting cooperative document production, rather than a generic approach to version management.

Likewise, a number of researchers have developed pragmatic solutions to the shortcomings of existing version management models. For example, Garg and Scattchi [8] have successfully integrated a hypertext system with RCS to control software documents. Similarly, Pendergast and Beranek [9] have addressed the problem in collaborative workstation design and Beaudouin-Lafon [10] has included a simple version model in a shared editing system.

In this paper, we wish to move beyond these specific approaches to consider a general model of versions that takes into account the fact versions will be used in a variety of different settings. We hope that this will be applicable across a range of cooperative systems. The model focuses on representing different contexts of user in a model of versioning suitable for the support of cooperative work. The starting point for our consideration is a reconsideration of what constitutes a version.

3. What is a version?

The word ‘version’ has many different meanings in information technology, software engineering and everyday language. Let’s start by looking at alternative definitions from the Shorter Oxford Dictionary. Surprisingly, perhaps, these more traditional definitions largely correspond to the common technical uses in computing.

Version... **1.** A rendering of some text... from one language into another... **2.** The particular form of a statement... given by one person or party; an account... embodying a particular point of view. **b.** A particular form or variant of something.... *The Shorter Oxford English Dictionary*

The first of these definitions (**1**) is specific and is used for example when talking about ‘versions’ of the Bible: the Authorised Version, the Revised Standard Version etc. Each ‘version’ attempts to reflect the original text(s); the difference is primarily one of presentation for different audiences and cultures. Of course, it isn’t as simple as this as presenting the material in different ways requires it to be interpreted and this, naturally, introduces differences in the content of the versions.

This is similar in many ways to ‘versions’ of software for different hardware/operating-system platforms. The version of Microsoft Word 6 for the Macintosh is as far as possible the same as Word 6 for Windows. In this case there are differences due, for example, to the different ways that object embedding is implemented on each platform. Although they are intended to be the same, their interpretation for a different platform has introduced some differences.

The second definition (**2**) is relevant when trying to understand the different perceptions people have of the same events in a cooperative situation. People’s interpretation of events is biased by their background and culture, by their state of mind when the events are observed, etc. It also has a more prosaic application when we consider different versions of a document as it is annotated, updated and edited by different people: Alan’s version of this paper, Tom’s version, Ian’s version. These are not like software versions for different platforms, but actually represent different viewpoints, which may at a later stage be reconciled.

The final definition (**2b**) ducks the issue, by referring to a ‘form or variant of something’. As the word ‘form’ is more elusive than ‘version’ we instead look at the definitions of ‘variant’:

Variant... **B. sb. 1.** A form or modification differing in some respect from other forms of the same thing.... **2.** A variation of the original work,... *The Shorter Oxford English Dictionary*

This is close to the meaning of the word ‘version’ as used in software engineering version management systems. In these contexts we have a linear or branched history of updated forms of an original program or document. Sometimes the versions are indexed by their place in the derivation tree (e.g. version 2.1).

Alternatively versions may be indexed by the times at which they were current, for example, where the versions are automatically kept by a backup facility.

Given these different meanings of the word ‘version’ should we be looking for a single model of versions or several models for each meaning? In fact, the meanings do have aspects in common. A single ‘thing’ can have more than one value associated with it. That is we want to model:

- *an entity with multiple values*

In fact, we have more common detail. The different values (‘versions’) that the entity has are often distinguished by the context in which the entity is encountered: people’s viewpoints (Alan’s version), hardware platform, or point in time. This leads to a working definition:

- *a version is the value of an entity in a context*

We will use this as the basis of our model of versions.

4. Basic Model

Look again at the statement “a version is the value of an entity in a context”. The fact that the same thing can have a different value in different contexts makes this a modal definition. So, we start with a definition of ‘worlds’ or contexts.

4.1 Worlds and contexts

The notion of a context is not one that is well-defined in cooperative systems. We don’t believe that a single definition of ‘context’ is possible as it depends on the type of application and the organisational setting where the application is used. Our model does not therefore fix what a context means and we work with its intuitive meaning.

We’ll label the set of contexts Λ and use λ for a particular context. An example of a context from Λ might be “Alan’s view of the world when typing at 8.54 pm on Saturday November the 26th”. In any particular context there is a set of observable features (from a set F) which have observed values from the set V . So a single ‘world’ is a function from F to V :

$$w \in W = (F \rightarrow V)$$

For example, consider a spreadsheet. Observable features would include the numeric values of each cell, the formulae, any global preferences etc. The set F is intended to be a ‘meaningless’ set of labels, merely there to distinguish observable features. However, they may well have names within a particular context and so in the case of the spreadsheet, we might well take F to include the set of cell addresses A1, B1 etc.

Note that observables need not be atomic nor independent. So, an observable in a context might be “the spreadsheet budget.wk1”, but also each of its cells and even cell ranges could be observables.

So, we have a set of contexts Λ and for each $\lambda \in \Lambda$, a world w_λ :

$$w_\lambda \in F \rightarrow V$$

Or in other words a function:

$$w: \Lambda \rightarrow W$$

The intent of what we have so far is to say that, in any context, certain things are observable. The set F is there merely to distinguish the ‘things’.

4.2 Entities

At this point we can introduce entities. So far, we can talk about the things that are observable in any particular context. However, the essence of a version is that several different things are in some sense considered to be the same thing. In the model, the entities capture this sameness.

We will call the set of entities E , but like F this is a labelling set which need not have meaning in itself. A cup for example is a cup whether or not it has a name. An entity is something which is identifiable in multiple contexts. That is, for some contexts there will be an observable which is the value of that entity in the context. The function ρ captures this association:

$$\rho : E \times \Lambda \rightarrow \mathbb{P}(F)$$

In some contexts, there may be more than one observable corresponding to a single entity. For example in figure 1 there are three files which are all early versions of this paper. This is why the function ρ associates a set of observable features with each entity. Also, some contexts may have no observables corresponding to the entity (for example there is no version of this paper in 1895) and so the set may be empty.

Figure 1: Simultaneous versions of a paper

A special case of this model is when we know that there is at most one version of the entity in any context. For example, this would be the case for physical entities. We can call the set of such single valued entities E_S (a subset of E). In such cases the definition function ρ can be restricted:

$$\rho_S : E_S \times \Lambda \rightarrow F$$

Note that this is a partial function still allowing contexts where there is no version. We will normally use the more general version in subsequent definitions. If we compose the function ρ and the world mapping w_λ , we get the function ver :

$$ver : E \times \Lambda \rightarrow \mathbb{P}(V)$$

$$\forall e \in E, \lambda \in \Lambda \bullet ver(e, \lambda) = \{ w_\lambda(f) \mid f \in \rho(e, \lambda) \}$$

This function ver corresponds exactly to the definition that a version is a value (V) of an entity (E) in a context (Λ).

4.3 Distinguishing things

Having defined entities in this model, you can see why we need both worlds and contexts. Suppose that our contexts are simply times and our worlds are 2 cell spreadsheets which only have two observable features called 'A' and 'B'. At time 0 the world is:

$$w_0 = \{ A \mapsto 3, B \mapsto 5 \}$$

Then the value of A is set to 5

$$w_1 = \{ A \mapsto 5, B \mapsto 5 \}$$

Then the value of B is set to 3

$$w_2 = \{ A \mapsto 5, B \mapsto 3 \}$$

Finally cells A and B are swapped.

$$w_3 = \{ A \mapsto 3, B \mapsto 5 \}$$

So the world w_0 is identical to w_3 but the context is different (times 0 and 3 respectively). Furthermore, imagine that the two cells represent scores in a board game. Alison is playing Brian. The changes at times 1 and 2 are due to points being won and lost by Alison and Brian respectively. However, at time 3 they move over to play on opposite sides of the board. To make it easier to score they swap the cells for the scores. The corresponding entities are: Alison's score and Brian's score.

We can trace the mapping ρ at each time:

$\rho(\text{Alison}, 0)$	=	A	$\rho(\text{Brian}, 0)$	=	B
$\rho(\text{Alison}, 1)$	=	A	$\rho(\text{Brian}, 1)$	=	B
$\rho(\text{Alison}, 2)$	=	A	$\rho(\text{Brian}, 2)$	=	B
$\rho(\text{Alison}, 3)$	=	B	$\rho(\text{Brian}, 3)$	=	A

See how at time 3, although the worlds are the same, the relationship between entities and observables (and consequently between entities and values) are different. Alison would not be pleased if the distinction were lost!

4.4 The current version

The fact that there are several possible values in any context captures the ambiguity that often exists (as in figure 1). However, it will often be the case that for any entity we have the idea of the ‘most up-to-date’ version of the entity, or the ‘official’ version of the entity. Note that ‘the most up-to-date’ version is different in different contexts. That is, it is itself an entity. In other words we have a partial function:

$$mud : E \rightarrow E_S$$

Note two things. First, this is partial function as the idea of a ‘most up-to-date’ version is not meaningful for all entities. Second, it relates entities to entities. It does not give you the current version of an entity in a specific context. However, this can be obtained from the above function:

$$curr-ver : E \times \Lambda \rightarrow V$$

$$\text{dom}(curr-ver) = \text{dom}(mud)$$

$$\forall e \in \text{dom}(mud), \lambda \in \Lambda \bullet curr-ver(e, \lambda) = w_\lambda(\rho(mud(e), \lambda))$$

This idea of the most up-to-date version is an example of a relationship between entities. We will return to the issue of relationships between entities, contexts, values and observables in section 6.

4.5 An Example

We can illustrate this with an example. Consider the following contexts - Alan’s PowerBook on 26th November 1994, the file ‘modal model’ is being edited, Tom’s desk on 3rd December, a piece of paper amongst many others. Two different contexts, two different things, but both versions of the same paper that you are now reading. Let’s see how the various sets and functions relate to this example.

On Tom’s desk are many different pieces of paper. They have no particular names or labels, but are still distinguishable. The set F is there precisely to enable us to talk about such anonymous observables. Imagine you go to Tom’s desk scribbling an arbitrary label on each piece of paper. In real life this is unnecessary, you can simply point to the one you mean. In a formal model the name is necessary to allow us to say which piece of paper we are talking about. The ‘values’ in the world w_{Tom} are the appearances of the various bits of paper.

Similarly one can label the different files, windows etc. on Alan’s PowerBook. In fact, for the files, one could use the file names as labels. In this case the values would simply be the file contents. Alternatively (and the more generally applicable approach) the name can be regarded as part of the ‘value’.

The fact that we can pick these two observable features in the two contexts and say that they are in some sense the same entity is recorded by the function ρ . So, if e is the entity corresponding to this paper:

$$\rho(e, \text{“Tom’s desk”}) = \text{“paper second to the top of the pile at the back”}$$

$$\rho(e, \text{“Alan’s Duo”}) = \text{“file ‘modal model’ in folder ‘version’ ”}$$

5. Labels

The use of labels is always a problem, so we’ll go back to the two sets of labels F and E and see why both are necessary.

5.1 Labels and names

In many cases there may be an obvious set of labels for the set of observable features F . In the case of a spreadsheet we can use the row–column addresses, in the case of computer files we can use the file name. However, we may also come across cases where there is no obvious labelling. Some additional labels have to be created to distinguish things such as the papers on Tom’s desk.

The purpose of labels is therefore to say ‘here is something’. The stage of giving it a meaningful name comes later (e.g., ‘the thing is a file called fred.c’). So, even where a sensible set of names exist it may be better to regard the name as an additional observable attribute of the thing (assuming it is observable!). This is rather like the issue of choosing keys in a database, whether one should use attributes of the entity, or whether one should invent arbitrary ‘surrogate’ keys.

This also helps us to keep track of changes in structure and names. For example, the file name ‘fred.c’ might seem like a good candidate for an entity name (indeed, that is precisely how many version control systems work), however, what happens if the file is renamed to ‘freda.c’? The same file has a different name in different contexts and could have either the same or different content! Similarly, when rows are inserted into a spreadsheet, a particular cell (regarded as a persistent entity) may easily change its address.

5.2 Why two sets of labels?

So, why are there two sets of labels, one for observable features and one for entities?

From the previous argument it would not be appropriate to use a file name or cell-address for an entity label as it may change. However, in the case when the labels in F are arbitrary, this is no problem. Why not simply use whatever labels we use for entities and regard the file name or cell address as another attribute of the entity in the context? Indeed, this is essentially what is happening in the *ver* function. Obviously including the extra level of indirection allows us to have meaningful feature names in a context, but given this is a semantic model, is this simply extra noise?

There are two good reasons for requiring the feature labels. The first is a matter of style. In a particular context we are aware, as noted above, that something is a thing, independent of its attributes (when you say ‘that is a frog’, you presuppose that the word ‘that’ is meaningful). The next level of understanding that the thing you are regarding is the same ‘thing’ as in a different context comes later. Indeed, such judgements may be hard to make (when the frog turns into a prince) and one may even want to discuss erroneous connections (I think that this is a version of document ‘X’, but it’s not).

The second reason is that there may not be a one to one mapping between entities and observables in a particular context. Some entities may not exist in some contexts, which is why all the mappings are partial. Furthermore, one observable may sometimes correspond to more than one entity. In the physical world, this would be somewhat odd, but if we have multiple perspectives this is quite natural. For example, consider the entities ‘the standard UK tax rate’ and ‘the lowest UK tax rate’. For many years, these not only had the same value, but were the same thing, now they are different things entirely.

Although these are all important considerations when looking at a model of what versions are, they may be ignored or simplified when building an actual version manager. So, when we start to formally define version managers in section 7, we will use context and entity labels which will be different from, but related to those used in the semantic model. For example, a program code version manager would very likely use file names as its entity names and consequently have problems when files are renamed. This is an acceptable simplification to make when implementing a system, but we need a richer model in order to discuss the essence of this issue.

5.3 Labels and variables

In a traditional temporal logic one may see formulae such as:

$$\Box x > 0 \Rightarrow \Diamond x > 10$$

This says (in linear temporal logic) that if it is ever true that x is greater than 0, then it will eventually, at some future time, be greater than 10. This formula relates different worlds (or contexts). It is usually implicit in the formula that the variable ‘ x ’ is referring to the same thing in each context (in our terms E and F coincide), but in fact this is irrelevant to the semantics: traditional model logics use proposition or variable names purely as carriers for predicates in each state.

In specification notations, the notion that the same name is ‘the same thing’ in different states of the world is usually explicit. This is often emphasised by primed names in formulae representing state change. However, where variable names are used to represent continuity between states there can be little structural dynamism and most specification notations, including those intended to represent dynamic object-oriented systems, make do with a predefined set of entities.

6. Relationships

We have already seen one example of a relationship between entities, where one entity is the ‘most up-to-date’ or ‘current’ version of another. Interesting relationships occur at all levels in the version model: between observables within a world or context, between entities and between contexts. In fact, the ‘most up-to-date’ relationship is perhaps most complex in that it is a meta-relationship. It only makes sense when we look at the world in terms of versions!

6.1 Relationships between features

The simplest relationships are those between observable features in a single world; for example comparisons between numeric features, such as ‘ $x > y$ ’, or geometric relationships between physical items, such as ‘inside of’. In a software engineering setting we may also have relationships such as ‘part of’ between a system and its components and ‘compiles to’ between a source program and its object file. In a CSCW setting, the relationships depend largely on the particular shared objects: folders and files have containment relationships; objects on shared drawing surfaces have geometric relationships; hypertext documents have links.

6.2 Representing relationships

As in standard database modelling there are two ways in which we can represent relationships within our formal model. One option is to explicitly extend the model of a world to include relationships between observables as well as the observable features themselves:

$$W' = (F \leftrightarrow V) \times (R \leftrightarrow \mathbb{P}(F^*))$$

That is, a world is a pair, the first part mapping features to values, the second mapping relationships (such as ‘part of’) to tuples of features that satisfy them. An example of such a world could be:

$$\begin{aligned} w' &= \langle \text{vals}, \text{rels} \rangle \\ \text{where } \text{vals} &= \{ (f1 \mapsto \text{'red cube'}), (f2 \mapsto \text{'blue pyramid'}) \} \\ \text{rels} &= \{ (\text{'on top of'} \mapsto \langle f2, f1 \rangle) \} \end{aligned}$$

The disadvantage of this approach is that we have another type of thing which may be versionable – we may want to regard the relationship between things as something with versions. The alternative is to model the relationships as part of the values of features. For example, part of the value of a folder is the set of files it contains (or, to be more abstract, their feature labels). If a relationship is not naturally part of a the value of a feature, but is observable, then it is reasonable to represent it as an observable feature in its own right. Using such a representation the previous example would simply be:

$$w = \{ (f1 \mapsto \text{'red cube'}), (f2 \mapsto \text{'blue pyramid'}), (f3 \mapsto \text{on_top_of}\langle f2, f1 \rangle) \}$$

Given both forms of representation have respectable pedigrees in data representation, we will adopt the latter which will simplify the modelling of version management itself. That is, we retain the original definition of a world from section 4.

6.3 Relationships between entities

Relationships between features are impermanent, they happen to hold in a particular context. In contrast, some relationships hold in all contexts. For example, the relationship between people ‘manager of’ may change with time, but the relationship ‘mother of’ never changes (surrogate parentage withstanding). The latter is a relationship between entities themselves rather than the particular values the entities take within a particular context. As these relationships are invariant they do not cause the same sorts of problems as those discussed in the previous section. We will not explicitly add them to the model we have given, but it is clearly not problematic to explicitly add relationships, functions and attributes of entities.

6.4 Relationships between contexts

Finally, we consider the relationships between contexts. The most obvious are to do with temporal or causal dependence. If contexts are simply the state of the world at different times, then we can order contexts by time, and if we measure time discretely, we may have a next and previous context. In a traditional software version control system, contexts are labelled with version numbers: 2.1, 3.1.2 etc. The relationship between these is usually one of derivation. Version 2.2 is derived from version 2.1. This sort of relationship is likely to be needed in most version management systems. This is also where issues such the use of branched or linear version histories belongs.

In the real(ish) world, causality is usually linked to actions, so that one context may be related to another by the action of “editing file ‘modal model’ ” This is central to most undo systems (a form of version management) which are indexed by actions.

In addition to these sequential or derivation relationships are those where two or more contexts are in a sense equivalent, for example Word 6 in the context of Macintosh or PC platforms. Under this broad heading we can include contexts such as “Tom’s version of ‘modal model’ at 2pm” and “Alan’s version of ‘modal model’ at 2pm”. Although the documents may be very different (if we were both editing sections), the relationship between the contexts is very important if we start to discuss the document by phone or some other synchronous channel.

To some extent, this example is a special case of contexts determined by several orthogonal attributes (in this case person and time). In such cases contexts sharing a particular attribute have an intrinsic relationship. However, time is not simply an attribute like any other, and may have to be dealt with specially.

7. Modelling Version Managers

A version manager is a system like SCCS which maintains information about and provides access to the versions of an entity. The entity names E are program source file names and the contexts Λ are version serial numbers (1,1.1,...2.1 etc.) The value set V is the file contents, that is lines of ASCII characters. Although SCCS makes use of particular methods (forward deltas) to store multiple versions efficiently, the model of the behaviour of SCCS can simply treat it as a mapping from version numbers to file contents.

So far we have talked about versionable entities. However, we have not modelled the explicit storage of multiple versions in a version manager. To do this, the version manager must itself hold some sort of modal model of the entities concerned. That is, the value of a version manager will contain a mapping of the form:

$$V \supset E' \times \Lambda' \rightarrow V'$$

Typically the version manager will not deal with all the contexts which arise in the real world. So, Λ' would be a subset or abstraction of Λ . Also, the entities in the version manager E' would only be some subset of the real entities E ; indeed In both cases, the names used by the users (Λ' and E') would typically be different from the arbitrary labels we use in the model itself (Λ and E).

Sometimes a version manager will deal with only one entity and so the above would become:

$$\Lambda' \rightarrow V$$

As this simplifies subsequent definitions we'll use this definition in the examples and further definitions below.

7.1 Felicity

Of course, the version manager could represent an entirely fictional version history. So, if one entity is supposed to represent versions of another entity, we expect some kind of felicity. To do this, we need to know the relationship between Λ and Λ' and then assert that the values that the version manager claims for an entity in context λ is precisely the value it did have in that context.

Let's formalise the above paragraph. We'll assume that the entity involved is single valued. Let $@(\lambda)$ be an actual context corresponding to a context label in the version manager (i.e., $@: \Lambda' \rightarrow \Lambda$). Then we could say that the entity e' is a felicitous version manager for e , given the mapping $@$, if:

$$\begin{aligned} \forall \lambda \in \Lambda : \quad & \forall \lambda' \in \text{dom } f : f(\lambda') = w_{@(\lambda')}(e) \\ \text{where } f = w_{\lambda}(e') \quad & \{ \text{N.B. } f : \Lambda' \rightarrow V \} \end{aligned}$$

In the case of SCCS, we would want to assert that the file `fred.c.s` was the version history of `fred.c`. The $@$ function could be the time at which the delta was taken. At a particular time (say 11am on 3rd Jan. 1995) we might see that `fred.c.s` said that version 1.2 has the value:

```
#include <stdio.h>
main()
{
    printf("fred");
}
```

If $@(v1.2)$ is 6pm on 17th Nov. 1994, then we would expect to find that at that time `fred.c` also had the above contents.

7.2 Reflexivity

In this section, we consider how the same modal model can be used to represent versions in different contexts and the version managers that maintain these versions. The simple nature of the model presented here allows us to consider a range of different extensions appropriate for its use in a number of application domains. One area of importance for cooperative applications is the modelling of temporal issues. These temporal issues are most important in the case of asynchronous applications where users may wish to view a number of shared items from the perspective of a cooperating user at a given time.

We have already discussed how the value of an observable in a particular context can itself represent versions of entities. Of course, this value will itself be associated with a particular entity – the version manager itself. As new deltas are stored in this entity it will take on new values in different contexts. Of course this could lead to infinite regress. We model versions of entities that include models of versions ...

This at first sounds impractical, but in fact does occur in practice. A tape backup system is a form of version manager. It stores versions of the complete state of a file system, usually indexed by time (that is its model of contexts Λ' is time). Now consider doing a backup when the file-system includes storage for a version manager such as SCCS. The backup tapes contain versions of the version manager entity which itself contains versions of files!

These reflexivity issues, although fun for a computer scientist, should not be apparent in any system that is intended for use by normal people. Happily, the reflexivity is trivial if the version management system is monotonic, that is if we only ever add to it. Consider again the case of the tape backup:

- At 2pm we do a backup of the complete file-system.
- At 3pm the file "modal model" is edited, a section on temporal databases is deleted and a new section on reflexivity is added.
- At 3:30 we commit the change into our version control system it records the new version as "modal_model!4.7" and stores it in its version database which is stored in a file "vmdb".

Now consider the state of the tape backup and the file system at 4pm. If we look at the file "modal model", both the backup and the file-system contain information that is not in the other. If we decided that the edits at 3pm had ruined the paper we would need the backup to restore the state of "modal model" at 2pm.

However, “vmdb” is different. Assuming the versions are timestamped, the file “vmdb” contains within itself its own history. Although the backup is necessary to protect against hardware failure, it is not necessary to restore the state of “vmdb”.

8. Conclusions

In this paper we have considered the development of a general model of versions appropriate for use in cooperative settings. The developed modal model considers entities in terms of their existence within particular contexts. Each context has an associated world view that maps the observable features of the world to particular values. A version in the model becomes the values of an entity within a given context.

The model does not restrict or specify what constitutes a context or a world. Interpretation of context could include time, user or part of work process. Thus we can model versions as they change over time; as they are seen by different users and as they alter as a cooperative activity is undertaken. Each of these different interpretations represents a specialisation of the general model represented here and allows a range of different cooperative applications to be developed.

We have also not discussed the means by which the model is presented to users. The development of appropriate presentation mechanisms will directly effect how the model is used in practice. For example, how aware are other users of the different contexts currently represented across an application and how are updates to these contexts propagated across a community of users. Should users be able to switch contexts to see entities as they once appeared to other users? Developing the answer to these questions requires us to construct a series of applications that realise the model presented here and to examine the utility and usage of the model in practice.

We are currently working on embedding this model in a experimental system to manage the versioning of WWW pages. We have chosen this example because it represents a real problem (‘how has this page changed from the last time I looked at it’, ‘where is the information on XX which Jean said was there’, etc.) and because we believe that Web-based applications are likely to constitute a large class of practical CSCW applications in future. We are exploring issues such as the presentation of the version model to readers of Web pages, navigation through this model and the capture and presentation of awareness information. All of these will be the subject of a separate paper.

9. References

1. Rochkind, M.J. (1975), ‘The Source Code Control System’, *IEEE Trans. on Software Engineering*, SE-1 (4), 255-65.
2. Tichy, W. (1982), ‘Design, Implementation and Evaluation of a Revision Control System’, *Proc. 6th Int. Conf. on Software Engineering*, Tokyo.
3. Lacroix, M., Roelants, D. and Waroquier, J.E., (1991), ‘Flexible Support for Cooperation in Software Development’, *Proc. 3rd Int. Workshop on Software Configuration Management*, Trondheim, June 1991.
4. Lie, A., Conradi, R., Didriksen, T.M. and Karlsson, E-A., (1989), ‘Change-oriented Versioning in a Software Engineering Database’, *Proc. 2nd Int. Workshop on Software Configuration Management*, Princeton, N.J., October 1989.
5. Bernard, Y. and Lavency, P., (1989), ‘A Process Oriented Approach to Configuration Management’, *Proc. 11th Int. Conf on Software Engineering*, 320-30.
6. Reps, T., Horowitz, S., and Prints, J., (1988), ‘Support for integrating program variants in an environment for programming in the large’, *Proc. 1st Int. Workshop on Software Configuration Management*, Grassau, January 1988.
7. Haake, A. and J.M. Haake (1993), ‘Take CoVer: Exploiting Version Support in Cooperative Systems’, *Proc. Conf. on Human Factors in Computing Systems (INTERCHI’93)*, Amsterdam.
8. Garg, P.K. and Scacchi, W., (1990), ‘A Hypertext System to Maintain Software Life-cycle Documents’, *IEEE Software*, 7 (3), 90-8.
9. Pendergast, M. and Beranek, M., (1991), ‘Coordination and control for collaborative workstation design’, *Proc. INTERACT’90*, Elsevier Science Publishers.
10. Beaudouin-Lafon, M. (1990), ‘Collaborative development of software’, in *Multi-user Interfaces and Applications*, eds. Gibbs, S. and Verrijn-Stuart, A., Elsevier Science Publishers.