Modelling Versions in Collaborative Work

Alan Dix

School of Computing, Staffordshire University, Stafford ST18 0DG, UK.
Email: A.J.Dix@soc.staffs.ac.uk

Tom Rodden and Ian Sommerville

Cooperative Systems Engineering Group, Department of Computing
Lancaster University, Lancaster LA1 4YR, UK. Email: {tam,is}@comp.lancs.ac.uk

Abstract

This paper addresses the problem of version management for cooperative systems. We first describe a basic version model – a domain model capturing the idea of a version and the relationships between versions, whether electronic or physical, implicitly or explicitly stored. This is a 'modal' model in that it takes into account the context of use of these versions. The model is then extended to model version managers, which are both versioned entities themselves and also include their own models of versions. Finally, we see how V-Web, a system for versioning web pages, fits within the framework of our model.

url for related work: `http//www.soc.staffs.ac.uk/~cmtajd/topics/versions/`

1. Introduction

Most prototype CSCW systems are based on a view of use that 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 that largely depend on a shared understanding of the work that 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. This mismatch is particularly evident in the lack of support for the evolution, management and co-ordination of the products of collaborative endeavours. We are not aware of any 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 only some of 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 capture only 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 a 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 makes 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. The paper concludes by describing V-Web, a versioning system for the World Wide Web, and using it as a simple example of the use of the model.

Note that, although the paper includes an abstract definition of a version manager, the intention is not to produce a framework or design pattern for the specification or implementation of version managers. Instead, we wish to capture formally the common elements of the various things we call 'version'. There are several extant specifications of configuration and version mangers in a software engineering context. However, by modelling the version manager first, they elide the issue of what versions are. Whereas this may be acceptable in a field where there are commonly understood concepts, this is not the case in less formal settings. We therefore believe that a clear understanding of versions themselves is a necessary precursor to effective communication about versions in collaborative work. To understand version management, we must first understand versions.

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 [17] and RCS [19] 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. 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, 12, 13]. 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. Much of the work that has occurred to date focuses on the merging of versions produced by different members of a group [15]. This has been concerned with merging independently made changes to software components. Other approaches have used the fine grain structure of files to reduce the impact of changes [18]. In general, the version models adopted by most existing version management systems all seek 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 from a control point of view to maintain an individualistic model. 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 [10]. 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 Scacchi [9] have successfully integrated a hypertext system with RCS to control software documents. Similarly, Pendergast and Beranek [14] have addressed the problem in collaborative workstation design and Beaudouin-Lafon [2] has included a simple version model in a shared editing system. Versioning can also be used as part of a strategy for managing conflicts in widely distributed or mobile situations where collaborative work on shared objects must continue asychronously at several sites [7, 8].

The importance of versioning has recently begun to be recognised within the WWW community. Reuter et al. [16] have demonstrated that Web browsers may be used as a front-end to a revision control system, although the objects being versioned in this system are not themselves web pages. Vitali and Durand [20] have proposed VTML, a markup language for storing document version information. This requires a parser to sit between the web server and the document store to process all documents on the fly. Work in progress at GMD on the CoopWWW project includes version management features [1]. The Working Group on Distributed Authoring and Versioning on the World Wide Web was set up in May 1996 to address problems created by the addition of write capability to HTTP. Internet standards in this area may eventually emerge.

Although we will exemplify our approach by describing a particular collaborative WWW versioning system, V-Web, the main aim of this paper is to move beyond specific approaches and to consider a general model of versions that takes into account the fact that 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. Our starting point 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 that, since 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, the interpretation for a different platform has introduced some differences.

The second definition (**2**) is relevant when we try to understand the different perceptions that 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 do not 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 8.54pm 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:

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

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, these 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 each of its cells and even cell ranges could also 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:

$$w: \Lambda \longrightarrow W$$

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 have been able to 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:

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

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.

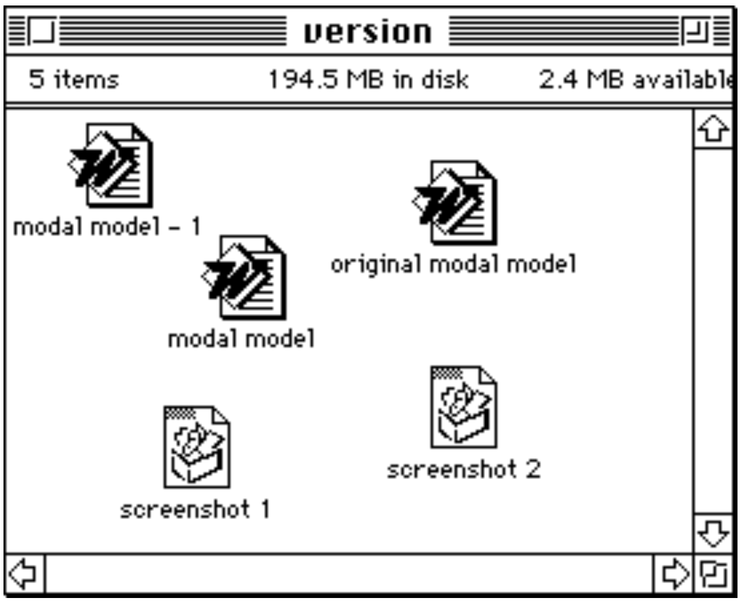


Figure 1: Simultaneous versions of a paper

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:

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

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*:

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

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

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:

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

Then the value of A is set to 5

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

Then the value of B is set to 3

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

Finally cells A and B are swapped.

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

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_S at each time stage:

ρ_S(Alison,0)	=	A	ρ_S(Brian,0)	=	B
ρ_S(Alison,1)	=	A	ρ_S(Brian,1)	=	B
ρ_S(Alison,2)	=	A	ρ_S(Brian,2)	=	B
ρ_S(Alison,3)	=	B	ρ_S(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) is 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:

$$\mathit{mud} : E \nrightarrow E_S$$

Note two things. First, this is a 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:

$$\mathit{curr\text{-}ver} : E \times \Lambda \nrightarrow V$$

$$\mathrm{dom}(\ \mathit{curr\text{-}ver}\ ) = \{\ (e, \lambda)\ |\ e \in \mathrm{dom}(\ \mathit{mud}\ ) \ \wedge\ (\mathit{mud}(e), \lambda) \in \mathrm{dom}(\rho_S)\ \}$$

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

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 1996, the file 'modal model' is being edited; Tom's desk on 3rd December, a piece of paper among 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 and scribble an arbitrary label on each piece of paper. In real life this is unnecessary, since 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 (an approach that is more generally applicable) 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:

ρ(e, "Tom's desk on 3rd Dec.")	=	{ "paper second to the top of the pile at the back" }
ρ(e, "Alan's Duo on 26th Nov.")	=	{ "file '`modal model`' in folder '`version`' " }

Let's embellish the story a little more. On the 1st of December the file on Alan's PowerBook was edited and saved with the new name '`modal model - 1`'. However, it is the printout of the first version that is on Tom's desk on 3rd December. So, from Alan's point of view there are two versions on 3rd December:

ρ(e, “Alan’s Duo on 3rd Dec.”) = { “file ‘`modal model`’ in folder ‘`version`’ ”,
“file ‘`modal model – 1`’ in folder ‘`version`’ ” }

Of these, the edited version is the most up to date for Alan.

ρ_S(*mud*(e), “Alan’s Duo on 3rd Dec.”) = “file ‘`modal model – 1`’ in folder ‘`version`’ ”

But for Tom, the most up to date is still his paper copy:

ρ_S(*mud*(e), “Tom’s desk on 3rd Dec.”) = “paper second to the top of the pile at the back”

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 exists 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 ‘`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 might change. However, in the case where 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 that 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 it 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 probably use file names as its entity names and consequently have problems when files were 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

The issue of names, values and persisitent entities is ubiquitous within computing, not least when considering variables in specification notations and programming languages. For example, 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 x 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 more 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.

In programming languages the pattern of relationships is more complex again. The same variable name can refer to different entities at different points in the program and different names may refer to the same thing. Furthermore, we cannot even associate an entity permanently with a location in memory; with dynamic memory mangement an object may move or even migrate or be replicated between different machines. Such behaviour is common in computer entities, but rarely found in real world entities outside the pages of fairy stories.

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.

In some cases, such as ‘x>y’, the relationship between features is inherited from the values of the features (in this case numeric comparison). However, structural relationships must be explicitly represented.

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:

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

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:

$w' = < vals, rels >$
where $vals = \{ (f1 \mapsto \text{‘red cube’}), (f2 \mapsto \text{‘blue pyramid’}) \}$
$rels = \{ (\text{‘on top of’} \mapsto <f2,f1>) \}$

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 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:

$$w = \{ (f1 \mapsto \text{‘red cube’}), (f2 \mapsto \text{‘blue pyramid’}), (f3 \mapsto \text{on\_top\_of}<f2,f1>) \}$$

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, between people the relationship ‘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 as the use of branched or linear version histories belong.

In the real(ish) world, causality is usually associated with actions, so that one context may be related to another by the action of "editing file `modal model`". In many undo systems (a form of version management) the states to which one can return are indexed by actions: 'paste', 'typing', etc. In contrast, version management systems often have no model of the actions or other relationships between versions with the exception of those within the user's comments on the new version.

In addition to these sequential or derivation relationships there are relationships 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. We will return to some issues of time in Section 7.3.

7. Modelling Version Managers

So far we have talked about versionable entities in their own right. This is important as a version manager only records some part of the complete set of versions that are and have been in existence. However, we now turn to version managers themselves – what they store (in abstract terms) and what it means for a version manager to faithfully represent the real world. The issue of version management is inevitably reflexive – a version manager stores information about versions of entities, but is itself an entity which changes and thus itself has versions. We will look briefly at some of the issues this raises.

7.1 What is stored

A version manager allows users to store and retrieve multiple versions of an entity. 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:

$$E' \times \Lambda' \nrightarrow V'$$

Typically the version manager will not deal with all the contexts that arise in the real world. So, Λ' would be a subset or abstraction of Λ. Also, the entities in the version manager E' would be only 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).

Consider a typical software version management system, like SCCS, which maintains information about and provides access to versions of program files. 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.

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

$$\Lambda' \to V'$$

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

In either case, the <u>value</u> of the version manager in a particular context is one of these context-to-value mappings. This means that when we consider the state of values of entities we must include these mapping values. That is:

$$V \supset (E' \times \Lambda' \nrightarrow V')$$

This is one of the rather odd ways in which the reflexive nature of this area manifests itself, an issue we will return to below. (Grammarians note the reflexive nature of the previous sentence and this one!)

7.2 Fidelity

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 fidelity. To do this, we need to know the relationship between Λ and Λ' and then assert that the value 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., $@: \Lambda' \nrightarrow \Lambda$). Then we could say that the entity e' is a faithful version manager for e, given the mapping @, if:

$$\forall\ \lambda \in \Lambda,\ \ \lambda' \in \mathrm{dom}\ w_\lambda(e') : \qquad w_\lambda(e')(\lambda') = w_{@(\lambda')}(e) \qquad \{\text{N.B. } w_\lambda(e') : \Lambda' \nrightarrow V\}$$

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 had the value:

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

If @(v1.2) is 6pm on 17th Nov. 1996, then we would expect to find that at that time fred.c also had the above contents.

Note that @ need not itself be constant over all contexts. In Section 8.4 we will see an example where @ must be considered as different in different contexts, that is for some kinds of version manager we must model it as: $@: \Lambda \times \Lambda' \nrightarrow \Lambda$.

7.3 Reflexivity

In this section, we are considering how the same modal model can be used to represent both 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 – that is, the version manager. 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 3:30pm 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’.

So we see that the reflexivity implied by versioning need not lead to infinite regress, as monotonic version managers contain within themselves a record of their own past states! Of course, if we require this, we do have to be careful about the precise information that is recorded. For example, if we are versioning files we must record not only the time the file was last updated, but also the time at which the version of the file was archived in the version manager. If this were not done, someone might update a file at 9am but forget to archive it until

11am. In the mean time at 10am someone else retrieves the (out of date) file from the version manager. Subsequently if things go wrong an audit trail of the problem would need to know what version was retrieved at 10am, which would not be the file dated 9am.

7.4 Naming contexts

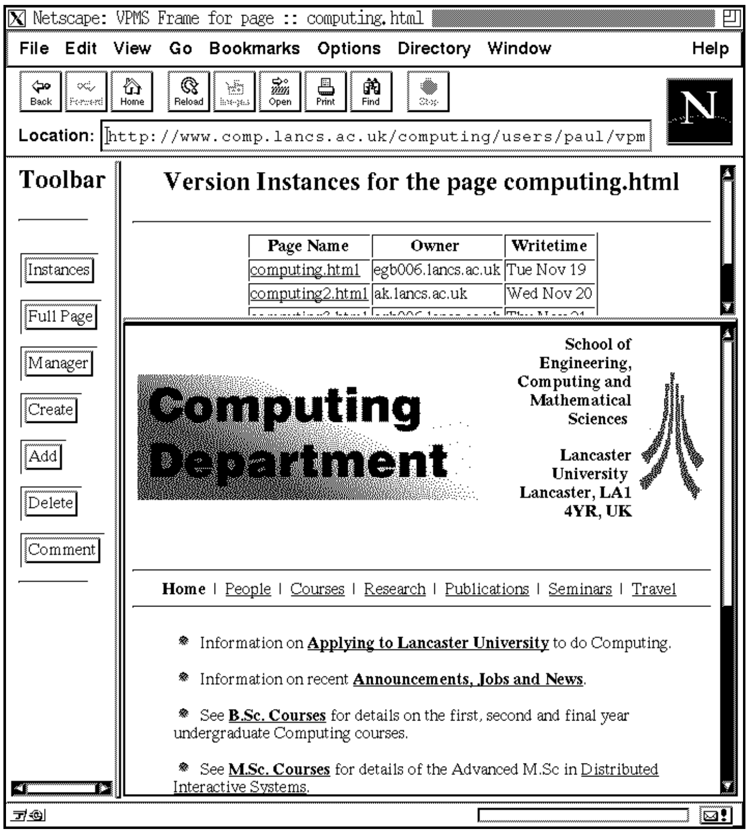
Version management is one of several issues where users and systems must look in on themselves, recording and talking about not just what is happening now, but also past states or traces of actions. Other examples are history mechanisms in hypertext and undo in any application [4, 5]. These reflexive features are typically hard to implement, hard to reason about and hard for people to understand. One of the reasons for the difficulty at the user interface is disagreement about what constitutes the history; for example, the undo button may undo the effect of the entire interaction with a dialogue box, or just the last button press [21].

Happily, traditional version management systems allow users to choose which versions to record, alleviating some of these problems. However, the way in which these versions are then *named* is often far from obvious to a naive user. In terms of the model this corresponds to the labelling of contexts Λ'. The example above used time as a way of labelling contexts, but often version managers use index numbers with different numbering schemes for each object and poor interfaces for browsing past versions.

So, when we look at a version manager using this model we should look carefully at not just the fidelity, but also the appropriateness of the context labels compared with users' natural ways of referring to contexts. This conflicts somewhat with our previous insistence that labels should be arbitrary. This advice was primarily aimed at the abstract domain model, but similar problems are also evident in implementations. Where context labels cannot be meaningful or where different users have different understandings of contexts we must look at the user interface to determine whether users can find or recognise past versions.

8. Applying the model – V-Web

Let us now illlustrate the application of this model by applying it to a simple version management system (V-Web) which we have developed for the World-Wide-Web [11]. We will briefly describe the operation of V-Web and then apply each of the features of the model. The relationship between V-Web and the model is not a strict derivation, but neither are they independent. V-Web was designed after the above model was formulated. It was thus influenced by some of the issues raised by the model, but also by many other driving factors, including the nature of the platform and a previous system which formed the basis of its construction. This meant that it was not inevitable that V-Web would fit the model and indeed in section 8.4 we will see how V-Web showed up a deficiency in the formulation of the fidelity operator @ introduced in Section 7.2.



http://www.comp.lancs.ac.uk/computing/users/paul/vpm

Version Instances for the page computing.html

Page Name	Owner	Writetime
computing.html	egb006.lancs.ac.uk	Tue Nov 19
computing2.html	ak.lancs.ac.uk	Wed Nov 20

Computing Department

School of Engineering, Computing and Mathematical Sciences

Lancaster University Lancaster, LA1 4YR, UK

Home | People | Courses | Research | Publications | Seminars | Travel

- Information on **Applying to Lancaster University** to do Computing.
- Information on recent **Announcements, Jobs and News**.
- See **B.Sc. Courses** for details on the first, second and final year undergraduate Computing courses.
- See **M.Sc. Courses** for details of the Advanced M.Sc in Distributed Interactive Systems.

Figure 2: A V-Web page

8.1 Versioning the web

The ease with which information can be updated on the web is one of its principal advantages over traditional paper-based information dissemination. However, it introduces its own problems, not least in the area of version management. This can be a problem for individuals, but is most significant when we consider collaborative aspects. First, there are potential problems for co-authored documents: avoiding overwriting others' work, keeping up to date with one's co-authors' work. Second, and perhaps easy to overlook, is the implicit cooperation between author and reader. Whereas a paper document is fixed and under the control of the reader, the web is a fickle medium where things seen today may be different or even non-existent tomorrow. This is particularly a problem for academic work where the ideas of archival documents and authoritative citations are predicated on the stability of media.

Figure 2 shows a V-Web page. The original web page has been replaced by a set of frames. The display frame (lower right) is the largest part, showing the original web page. Above is the version information frame which lists all the versions of the page. With this, users can see who has produced versions and also select previous versions to view in the display frame. Finally, there is the toolbar on the left-hand side, which gives access to different aspects of V-Web's functionality including the ability to add or delete versions and to add comments to particular versions.

The V-Web interface does not support editing directly, but allows authors to upload and download versions of a document, editing them locally using their own page generation software. CGI scripts are used to implement the upload of new versions and similar operations. We also provide some support for co-authoring where groups cooperate on the development of web pages. This is implemented through an access control mechanism which allows people within groups to be aware of the work being done by other group members and which allows for automatic notification when new versions become available. Registered authors may also have 'version sets', enabling registered users to have private versions for in-progress work. Figure 3 illustrates the situation where there are several users with different version sets. Steve is unregistered and sees the public version set with `page1.html` as the default; Paul has registered and has set his default to `page2.html`; Andy has a number of private versions with his default set to `newpage4.html`.

The V-Web page replaces the original unversioned page at the same URL so that any existing links to the original page are not disrupted. Navigation through any of these existing links will obtain the V-Web page showing the most recent version of the document. However, it is also possible to add a link to a particular document version. In the case of academic work it will be clear which version of a web page is being referred to, but it is also possible to find more up-to-date versions.

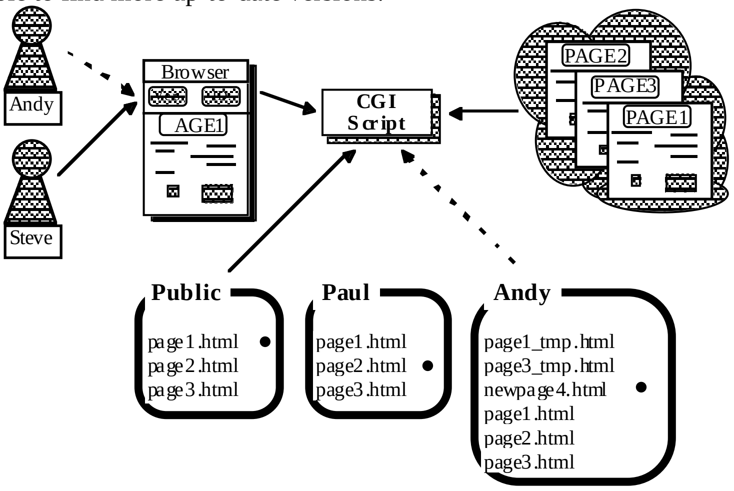


Figure 3: Supporting multiple authors

8.2 Domain model

We now start to model V-Web using the modal model. We begin by modelling the domain. For this exercise we will ignore copies of web pages held on local machines and only consider those stored on web servers. This reflects the fact that the most significant update action is the upload of a new version of a web page to the server. A richer model could also consider local copies of files, but since V-Web is server based that is not necessary to deal with this example. Thus the world at any moment can be considered to be simply a mapping from URLs to their content:

$$W = \text{URL} \nrightarrow \text{URL-content}$$

That is, the observables, F, are URL addresses and the values, V, are the contents of URLs. The URL content may be a simple web page, or may be a CGI script or other complex object. V-Web only manages normal HTML web pages; however, V-Web is itself built using CGI scripting. There will be many other kinds of web object, but for the purposes of this paper we will assume that the only two kinds of URL content are raw web pages and V-Web pages:

$$\text{URL-content} = \text{Web-pages} \cup \text{VW-pages}$$

The contexts of interest are defined by different viewpoints (public or authors) and the time the pages are viewed:

$$\Lambda = \text{Time} \times \text{ViewPt}$$

The mapping from Λ to W gives the subjective history of the world in different contexts. For now we shall assume that this is the same from all viewpoints:

$$\forall\ t \in \text{Time}\ \forall\ \lambda_1=(t,v_1), \lambda_2=(t,v_2) \in \Lambda \qquad w_{\lambda_1} = w_{\lambda_2}$$

Entities are always difficult to describe as one naturally tends to use attributes to describe them. A typical entity for V-Web would be the ‘ECSCW97 home page’. A particular web page that is (for whatever reason) regarded as being the ‘ECSCW97 home page’ may or may not have this string as its title. The entity set always becomes more concrete when its semantics are defined using ρ. Let’s consider the scenario which may have led to the situation on November 21st depicted in Figure 3.

First consider the public view. A permanent URL will have been given in publicity material:

$\forall\ t \in$ Time ρ(‘ECSCW97 home page’,(t,Public))
= { ‘`http://www.ecscw97/home.html`’ }

Of course, although the URL is fixed, the content may vary with time. If this page is maintained by V-Web, this will be the location which V-Web manages showing an appropriate physical page (on Nov 21st, the contents of ‘`page1.html`’). On a non-V-Web managed page, the content may change by editing, overwriting or the use of aliases.

On November 21st, Paul and Andy are working on different versions of the page (hopefully to be successfully merged later!). They thus have different viewpoints:

ρ(‘ECSCW97 home page’,(Nov 21,Paul))
= { ‘`http://www.ecscw97/page2.html`’ }
ρ(‘ECSCW97 home page’,(Nov 21,Andy))
= { ‘`http://www.ecscw97/newpage4.html`’ }

A few days earlier, Andy was working on a different copy of the file, which to him, at that time, was the ECSCW97 home page:

ρ(‘ECSCW97 home page’,(Nov 19,Andy))
= { ‘`http://www.ecscw97/page1_tmp.html`’ }

Although these viewpoints are labelled by individual authors, it is quite possible that authors may be aware of each other’s versions and very likely that all will be aware of the current public version.

8.3 Labels, names and links

Recall the importance of having entity labels that are not ‘names’. This is particularly clear here. If URLs were used as entity labels we would have no way of describing what happens when a page or server is moved. This is precisely what happens when a link breaks. However, one still has the idea that the ECSCW97 home page is out there somewhere, but it’s not where it used to be! Note also that different people may regard different URLs as denoting the same abstract web page. As well as being due to updates in progress (as in the example here) this may also arise because of mirrors. Thus the URL ought to be seen as an attribute of the entity in a particular context, not as defining the entity. This is part of the reason for the (slow) movements towards uniform resource names (URNs), which will give a web page a unique name no matter where it is physically located. These would be a better candidate as an entity label, although it would be still be possible to swap the content of a URN for something entirely different!

We have used URLs as feature labels. This is because they are unique in any context. An alternative representation would have been to use arbitrary labels and regard the URL as another attribute of the observable value.

In section 6 we discussed different kinds of relationship between values and entities and the way they can be represented. Perhaps the most important relationship between web pages is the clickable links. These may of course change as the pages are edited and are part of the value (HTML text). Note that the intention of a link is normally to point to a particular entity (say the ECSCW97 home page), but if the page moves the link becomes broken. Although some web servers aid the user by providing redirection the fundamental problem is that the abstract idea of ‘the ECSCW97 home page’ is a conceptual entity, whereas URLs are in the end storage addresses.

8.4 Version manager model

Each V-Web page manages a single (abstract) web page, hence we can use the simplified form for a version manager from section 7:

$$\text{VW-page} \quad = \quad \Lambda' \rightarrow V'$$

A context is simply a viewpoint (public or an author) and an index saying which version is being considered.

$$\Lambda' \quad = \quad \text{Index} \times \text{ViewPt}'$$

The index is either ‘default’, the current default version, or an integer representing the nth version in the set.

Note that the V-Web does not store by time, but by the order in which new versions are added. However, the time of update and the author of the update are both stored as attributes of the version. Hence the value stored for each version consists of the page contents, the corresponding URL, the time of update and the author.

$$V' \quad = \quad \text{Web-page} \times \text{URL} \times \text{Time} \times \text{Author}$$

In section 7.4 we noted that difficulties may arise if the context labelling set in the version manager, Λ’, is not understood by the users. In V-Web, as in many versioning systems, the versions are simply ordered in a list. However, the update time and author information are also made readily available, helping to identify required versions. In addition, it is substantially easier to browse through previous versions than in many version control systems!

In order to determine the fidelity of the model we must define the @ mapping between the V-Web context labels Λ’ and the domain model labels Λ. Note first that this mapping itself depends on the context. If a version is deleted from the system, the V-Web labels will change, so the mapping between Λ’ and Λ must also change. This means that strictly @ should be a function of Λ just as the world w_λ is. In addition, the information needed to determine the model lies within the state of the version manager itself:

$$\begin{aligned} &@_\lambda(\lambda') \quad = \quad (\, t, v \,) \\ &\textbf{where} \quad \lambda' \; = \; (\, n, v \,) \\ &\qquad\quad \text{VW-page}(\lambda') \quad = \quad (\, \text{w-page}, \text{url}, t, a \,) \end{aligned}$$

Recall that in Section 7.2 we originally defined @ in a context independent fashion. This is only true when the labelling scheme used by the version manager is fixed forever (once new versions are added). This is true of many version management systems, but was an unstated and unintended assumption. It was only when V-Web was cast within the framework that we discovered the problem. On the one hand, a formal model allows us to understand issues in the abstract and may bring to light important issues. However, it may also incorporate hidden assumptions. It is thus crucial that any formal (or informal) framework is tested by applying it to a diverse range of concrete examples.

There remains another difficult (ultimately intractable) problem. If we interpret the labels in Λ as being person X’s idea of the web page at time t, then how do we know what person X thinks! As in any user interface specification one can never entirely capture the user’s intentions and knowledge; however, this at least gives us a guideline as to what questions to ask when evaluating the chosen labelling and interface.

8.5 Notification and caching

The domain model we used in section 8.2 assumed that each person sees the same URL content. However, due to caching by browsers or web proxies, the same URL typically may appear differently to different people. Suppose Andy is viewing a particular page. Paul edits and updates it and then views the result. Paul will see the updated page, but even if Andy is looking at the same URL he will continue to see the old version until the browser reloads the page. In terms of the domain model, we may have at some time t:

$$w_{(t,v_1)} \neq w_{(t,v_2)}$$

To study problems due to lags between updates occurring and other authors becoming aware of the changes we need additional viewpoints such as ‘Paul’s idea of what Andy’s current version is’, or ‘Andy’s idea of what the current Public version is’. Although we cannot pursue this here it is would be necessary when considering notification services.

In fact, part of the ongoing development of V-Web is to add email notification of updates by other authors. The implementation of this is quite simple, but designing appropriate user customisation and notification rates is more problematic. There is a limit to how many email messages you want telling you that web pages have been changed!

9. 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 during cooperative activity. Each of these 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 general means by which the model is presented to users. The presentation mechanisms directly affect 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?

The V-Web system has been developed in part to help investigate such issues. The Web was chosen as an example domain both because it represents a real problem (‘how has this page changed since 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. As we saw, V-Web can be examined using the modal modelling framework, helping to highlight critical features of it and the web over which it is built.

Finally, we believe that the principal contribution of this paper is in establishing a framework within which we can discuss the meaning of versions. In fact, the issues raised seem to range far further than version management itself. Indeed, at an early presentation of this work at the 1996 FAHCI Conference [6], the discussion after the paper covered not only computing itself, but also mythology and theology!

Acknowledgements

This work was supported by EPSRC grant GR/J08560. The V-Web system was produced by Andy Kirby and Paul Rayson. Thanks also to the reviewers for their detailed and helpful remarks.

References

1. Appelt, W. 1996, CoopWWW - Interoperable Tools for Cooperation Support using the World-Wide Web. in *Proc. ERCIM Workshop "CSCW and the Web"*. Sankt Augustin, Germany: Arbeitspapiere der GMD 984, GMD/FIT. p. 91–94.
2. Beaudouin-Lafon, M., 1990, Collaborative development of software, in *Multi-User Interfaces and Applications*, S. Gibbs and A. Verrijn-Stuart, Editors. Elsevier Science Publishers:
3. Bernard, Y. and P. Lavency. 1989, A Process Oriented Approach to Configuration Management. in *Proc. 11th Int. Conf on Software Engineering*. IEEE Press. p. 320–330.
4. Dix, A. and R. Mancini, 1997, Specifying history and backtracking mechanisms, in *Formal Methods in Human–Computer Interaction*, P. Palanque and F. Paterno, Editors. (in press) Springer-Verlag: London.
5. Dix, A., R. Mancini, and S. Levialdi. 1997, Communication, action and history. in *Proceedings of CHI'97*. Atlanta, USA: ACM Press. p. 542–543.
6. Dix, A., T. Rodden, and I. Sommerville. 1996, A Modal Model of Versions. in *FAHCI – Formal Aspects of the Human Computer Interface*. Sheffield: Springer Verlag, Electronic Workshops in Computing.
7. Dix, A.J., 1995, Cooperation without (reliable) Communication: Interfaces for Mobile Applications. *Distributed Systems Engineering*. **2**(3): p. 171–181.
8. Dix, A.J. and V.C. Miles, 1992, Version control for asynchronous group work. YCS 181, Department of Computer Science, University of York, (Poster presentation HCI'92: People and Computers VII).
9. Garg, P.K. and W. Scacchi, 1990, A Hypertext System to Maintain Software Life-cycle Documents. *IEEE Software*. **7**(3): p. 90—8.
10. Haake, A. and J.M. Haake. 1993, Take CoVer: Exploiting Version Support in Cooperative Systems. in *Proc. Conf. on Human Factors in Computing Systems (INTERCHI'93)*. Amsterdam: ACM Press. p. 406–413.
11. Kirby, A., P. Rayson, T. Rodden, I. Sommerville, and A. Dix. 1997, Versioning the Web. in *7th International Workshop on Software Configuration Management*. Boston, USA: . p. 163–173.
12. Lacroix, M., D. Roelants, and J.E. Waroquier. 1991, Flexible Support for Cooperation in Software Development. in *Proc. 3rd Int. Workshop on Software Configuration Management*. Trondheim: Springer-Verlag.
13. Lie, A., R. Conradi, T.M. Didriksen, and E.-A. Karlsson. 1989, Change-oriented Versioning in a Software Engineering Database. in *Proc. 2nd Int. Workshop on Software Configuration Management*. Princeton, N.J.: ACM Press. p. 56–65.
14. Pendergast, M. and M. Beranek. 1991, Coordination and control for collaborative workstation design. in *Proc. INTERACT'90*. Elsevier Science Publishers.
15. Reps, T., S. Horowitz, and J. Prints. 1988, Support for integrating program variants in an environment for programming in the large. in *Proc. 1st Int. Workshop on Software Configuration Management*. Grassau: ACM Press. p. 197–216.
16. Reuter, J., S.U. Hänßgen, J.J. Hunt, and W.F. Tichy. 1996, Distributed Revision Control via the World Wide Web. in *6th Int. Workshop on Software Configuration Management*. Berlin: Springer. p. 166–174.
17. Rochkind, M.J., 1975, The Source Code Control System. *IEEE Trans. on Software Engineering*. **SE-1**(4): p. 255–65.
18. Sachweh, S. and W. Schäfer. 1995, Version management for tightly integrated software engineering environments. in *Proc. 7th Int. Conf. on Software Engineering Environments*. IEEE Computer Society Press. p. 21–31.
19. Tichy, W.F., 1985, RCS – a system for version control. *Software Practice and Experience*. **15**(7): p. 637–654.
20. Vitali, F. and D.G. Durand. 1995, Using versioning to support collaboration on the WWW. in *Fourth World Wide Web Conference*.
    Electronic publication at `http://www.w3.org/pub/Conferences/WWW4/Papers/190/`.
21. Wright, P., A. Monk, and M. Harrison, 1992, State, display and undo: a study of consistency in display based interaction. University of York.