Working Paper Series
ISSN 1170-487X

# Collaborative, Integrated Software Development with Multiple Views

**by John C Grundy, John G Hosking, Warwick B Mugridge, Robert W Armor**

Working Paper 94/6
May, 1994

# Collaborative, Integrated Software Development with Multiple Views

John C. Grundy*, Warwick B. Mugridge†, John G. Hosking† and Robert W. Amor†

## Abstract

This paper presents a new model for supporting collaborative, integrated software development utilising multiple textual and graphical views. Views can be asynchronously edited by different developers using separate versions. Versions can be incrementally merged, with merge conflicts detected and presented. Synchronous editing of views by different developers is also supported. View updates are broadcast to other users and are incrementally incorporated as required in their alternative versions. The new model is illustrated by its use in a software development environment for an object-oriented language.

## 1. Introduction

Software systems are growing ever larger and more complex. Two related approaches to managing this complexity are integrated software development environments (ISDEs) and collaborative programming environments.

ISDEs support multiple tools which assist in the development of complex software systems. Multi-view editing support in these environments allows developers to work with software components at different levels of abstraction and using different representations [Meyers 91]. For example, in an ISDE for object-oriented programming, analysis and design views might support graphical construction and representation of the high-level aspects of a program, while textual views might support detailed implementation of class interfaces and methods. As these views share some information (class and feature names, types and arguments, class relationships), consistency management is required to keep all of the views consistent when one view is edited.

Large software systems require the collaboration of multiple developers, with each developer viewing and manipulating information shared with others [Nascimento and Dollimore 93, Magnusson et al 93]. Support for collaboration can be provided by two types of tool: version control systems, which allow alternative designs to be created and merged asynchronously; and synchronous editors, which allow concurrent manipulation of a system by two or more collaborators [Magnusson et al 93]. Ideally an environment should support both synchronous and asynchronous modes for all types of software system components, allowing developers to switch between modes as required.

Our recent research has been concerned with the development of MViews, a framework for constructing multi-view ISDEs [Grundy and Hosking 93], together with SPE, an ISDE for an object-oriented language [Grundy et al 94]. In this paper we describe the extension of MViews to include support for the development of collaborative multi-view ISDE's. Extending the MViews multi-view approach to support collaborative development required support the following additional functionality:

- A version control system that deals with software components at a suitable level of granularity, and which allows for asynchronous and synchronous editing of multiple views.
- The ability to retrieve or regenerate earlier versions.
- Facilities for merging two or more completed versions that have been developed asynchronously by one or more developers.
- Facilities for incremental merging of two or more incomplete versions that are being developed in parallel by multiple developers and which are related by synchronous editing.
- A means of detecting merge conflicts that are immediately apparent from the changes concerned. For examples, two changes that operate on a single component in an inconsistent manner.
- A means of detecting indirect merge conflicts that can only be determined by semantic processing.
- A means of storing and distributing multiple versions, with appropriate identification.
- A means of traversing historical information associated with different versions.
- Facilities for developers to provide information about a high-level change and relate it to the set

* Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand. jgrundy@waikato.ac.nz

† Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand.

of low-level changes that will be made or have been made.

The remainder of this paper discusses how these needs are met in the new ISDE environment. Section 2 discusses related research. Section 3 describes the single-user ISDE that we have extended in this research. Section 4 describes the basic version control mechanism, while Sections 5 and 6 show how this mechanism can be used to support asynchronous and synchronous collaboration. Section 7 discusses implementation issues. Section 8 summarises the contributions of this research and outlines possible future research directions.

## 2. Related Research

PECAN [Reiss 85] and Garden [Reiss 86] support software development via multiple textual and graphical views. This allows developers to view software systems using different representations and at different levels of abstraction. Garden allows programmers to share software components and their views via an object-oriented database; however, it does not support version merging or collaborative view editing.

Tools such as SCCS [Roekind 75] support version control for text source code files. SCCS stores the differences between two versions of a text file as "deltas", which allows different versions to be regenerated and merged asynchronously. This technique does not work so well for more structured information, such as diagrams. In addition, the version merging process can be tedious and error-prone. FIELD environments [Reiss 90] allow Unix tools to be integrated by selective broadcasting of changes that occur in other tools. FIELD supports version control with SCCS and does not support collaborative view editing.

Collaborative document editors support synchronous, collaborative work on a shared document by a group of people [Ellis et al 91]. They do not usually support asynchronous editing and version control, as the multiple users are assumed to be working on the same document. For example, OpenDoc [OpenDoc 93] supports synchronous editing and versions via its object base, although automatic support for version merging is not provided.

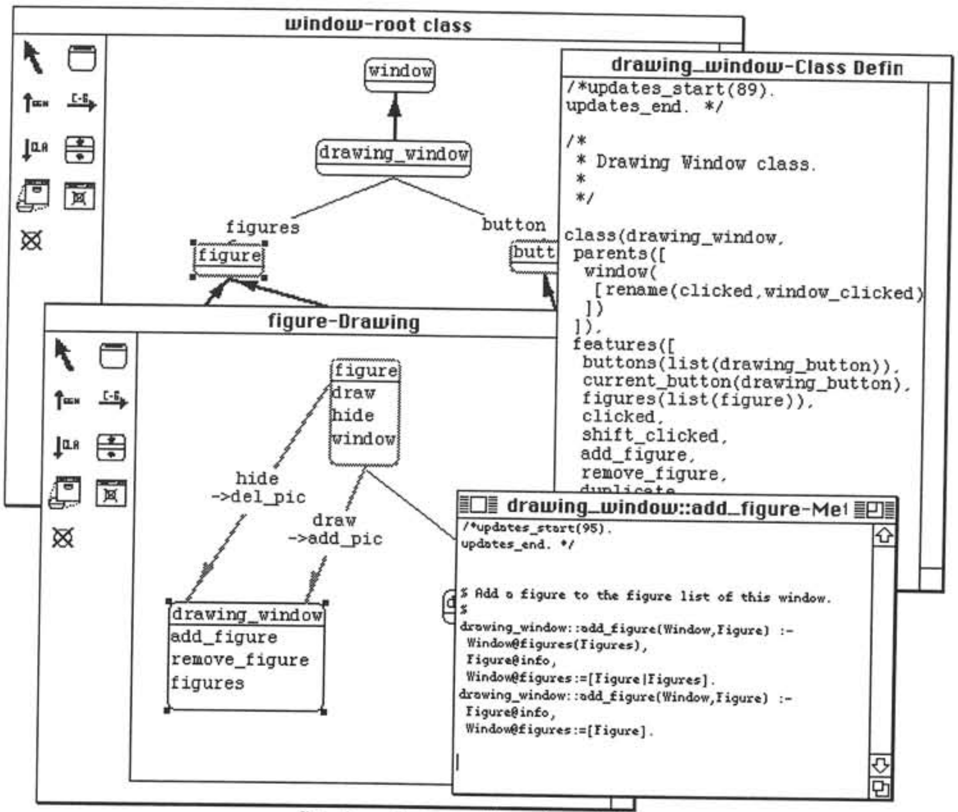
Mercury [Kaiser et al 87] extends the Cornell Program Synthesizer [Reps and Teitelbaum 87] to support a restricted form of collaborative programming. Changes to module interfaces are broadcast to other users. Mercury does not support versioning of program modules and only a single, textual view of a module is supported. Nascimento and Dollimore [93] propose a programming environment which supports (manual) version control for multiple programmers working on a shared Smalltalk program. Their approach does not support synchronous editing and multiple views. Mjølner/ORM environments [Magnusson et al 93] use a fine-grained version control system which supports both asynchronous and semi-synchronous editing. Mjølner environments also support "active diffs" which identify aspects of a program which have been changed by other developers. These environments provide only a single, textual, structure-edited view of program code.

Dora environments [Ratcliffe et al 92] provide multiple textual and graphical views of software development. Dora, however, supports neither version control nor the propagation of changes that can not be directly applied to a view. Thus it does not support "fuzzy" view updates between analysis, design and implementation views, and makes reconciliation of these views dependent on programmers remembering old updates.

Before discussing the details of the new framework and programming environment, we introduce the single-user versions that have been extended for collaboration.

## 3. MViews and the Snart Programming Environment

SPE (Snart Programming Environment) provides multiple textual and graphical views for constructing programs in Snart, an object-oriented Prolog [Grundy et al 94]. SPE supports full consistency management between all view types; changes to one view are always reflected in other views that share the updated information. For example, fig. 1. shows a screen dump from SPE showing two graphical views (one for object-oriented analysis and one for design), and two textual views (a class interface and a method implementation).



```
/*updates_start(89).
updates_end. */

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
 parents([
  window(
   [rename(clicked,window_clicked)
  ])
 ]),
 features([
  buttons(list(drawing_button)),
  current_button(drawing_button),
  figures(list(figure)),
  clicked,
  shift_clicked,
  add_figure,
  remove_figure,
```

```
/*updates_start(95).
updates_end. */


$ Add a figure to the figure list of this window.
$
drawing_window::add_figure(Window,Figure) :-
 Window@figures(Figures),
 Figure@info,
 Window@figures:=[Figure|Figures].
drawing_window::add_figure(Window,Figure) :-
 Figure@info,
 Window@figures:=[Figure].
```

fig. 1. A screen dump from SPE.

Graphical views are kept consistent when other views are modified by directly updating affected icons, expanding new icons into the view, shading icons to indicate the software components they render have been deleted, or storing with affected icons a sequence of change descriptions that can be viewed in a dialogue. Consistency management for textual views involves expanding descriptions of changes into the view's text in a special header annotation. Some of the expanded changes can be automatically applied by SPE to update the view's text. Other changes represent "fuzzy" changes affecting the view (eg a design level change propagated to an implementation view) which must be implemented manually as they require creative input from the programmer. To aid developers in determining the consequence to other views of a change, SPE supports a rich set of view navigation facilities, utilising hypertext techniques.

SPE is implemented by using a framework of Snart classes, based on the MViews model [Grundy and Hosking 93]. MViews supports the definition of new ISDEs by providing a general model for defining software system data structures and views, with a flexible mechanism for propagating changes between software components and their views to keep them consistent. With MViews, ISDE data is described as *components* with *attributes*, linked by a variety of *relationships*. Relationships behave as components and can thus participate in further relationships.

To support multiple views, a base layer defines base software system components (base components). A subset layer defines view components, which are partial views of the base components. These view components are grouped into views, which are rendered in either textual or graphical forms by a display layer. Developers update view components by direct manipulation of graphical views and free-editing of textual views in the display layer.

When a component is updated, a change description is generated (called an *update record).* For example *update_attribute(comp1, attr1, oldvalue, newvalue)* describes the change of value of attribute *attr1* of component *comp1* from *oldvalue* to *newvalue*. Update records are propagated to all components connected to the updated component which are dependent upon changes to its state (called *dependents*). Dependents interpret the update description and possibly modify their own state, producing further update records. This propagation process continues until all components affected by the original change have updated their state appropriately.

The update record mechanism is used to support a diverse range of software development environment facilities, including: semantic value recalculation; multiple views of a component; flexible, bi-directional textual and graphical view consistency management; a generic undo/redo mechanism; and component "update history" information.

The MViews framework also provides abstractions for building view editors and defining component view rendering and interaction. New environments are implemented by specialising MViews framework classes to define new components and relationships. A persistent object store is used to store component and view data.

Neither MViews nor SPE support collaborative, multi-user software development. We have extended MViews to produce C-MViews (Collaborative Multiple Views). C-MViews supports a flexible component and view version control mechanism and both asynchronous and synchronous view editing for collaborative software development. We have implemented a prototype of C-MViews and used this to construct C-SPE, which supports collaborative object-oriented software development.

In the following section we describe the basic versioning mechanism of the C-MViews model. This is based on storing and manipulating MViews-style update records. In Section 5, we show how this mechanism can be used to support asynchronous version editing and merging. In Section 6 we show how distributing update records between multiple users can be used to support synchronous merging.

## 4. Version Control

### 4.1 Component Version Control

Software system components usually have a natural hierarchy, with some components being "composed of" other sub-components. For example, an object-oriented program in SPE is made up of several class frameworks (or patterns); a framework is composed of several classes and class relationships; and a class is composed of various features and inter-class relationships.

There are several possible approaches to managing versioning of such hierarchical software systems. One approach is to create a new version number for the whole system whenever any (set of) changes is to be made to the system. A finer grained approach is taken by SCCS. Here, individual version numbers can be maintained for one level of the component hierarchy; thus each framework could have its own version number. The main difficulty with this approach has been that independent, usually manual, systems are needed to relate the framework versions that form a system version. A more general approach is to allow individual versions for any component in the hierarchy, with a configuration management tool able to construct a system version from versions at multiple levels of the hierarchy.

Rather than opting for one of these approaches, C-MViews aims to allow any of them to be implemented by providing a tailorable low-level versioning mechanism based on stored sequences of update records called *version records*. Specialisations of C-MViews, such as C-SPE use this low level mechanism to implement appropriate higher-level mechanisms.

### 4.2 Version Records

Version records are associated with C-MViews components. Creating a new version of a versionable component involves creation of a new version record.

Version records contain a record of changes made to that component since the previous version. These can include: update records associated with changes to the component itself; update records associated to changes to subcomponents; and changes to the configuration (ie the version numbers) of subcomponents. What is actually included in a version record depends on the versioning scheme implemented by the C-MViews specialisation.

For example, fig. 2 shows how version records are used in C-SPE's version control. C-SPE adopts a component-level versioning approach where each component in the component hierarchy has multiple versions with associated version records.

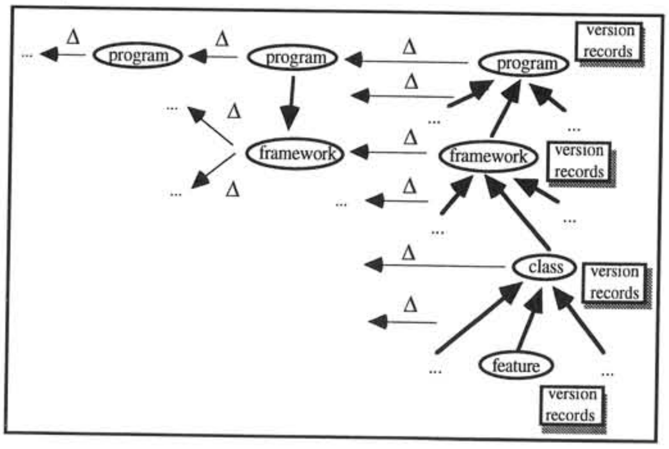


fig. 2. The general structure of C-SPE versions.

The most recent version of a C-SPE component is either frozen, and hence closed to change, or open, and hence able to be further modified. The version records describe changes made to a component: feature deltas (in the form of update records) describe changes between one feature version and the next; class deltas describe changes to a class component and its relationships and so on. Higher-level component version records also describe the particular configuration of their sub-components: class versions describe which versions of features and relationships owned by the class are used for a particular version of the class; framework version records describe which class versions are used for each version of a framework; and so on.

Each time a new C-SPE component version is created, such as a new feature implementation, a new version record for that component must be created to record the modifications to the component. The component's parent, eg the feature's class, in the aggregation structure must be notified of this new subcomponent version. If the parent's version is closed to change (ie it is frozen) a new version of the parent is created, and with it a new version record. The latest parent version record then records the changed subcomponent configuration. This propagation of configuration change is transitive up the aggregation hierarchy: if the change causes a new version of the class to be created, this in turn informs the framework and so on.

Other versioning schemes can also be supported using component version records. For example, the global version number method described in Section 5.1, can be implemented by simultaneously creating new (open) versions for every component, all labelled with the global version number. Subcomponent configuration information does not need to be maintained in this scheme, as all subcomponents will have the same version number.

For small types of subcomponent, it may be sensible to have version records only associated with their parents in the aggregation structure. For example, in C-SPE, rather than having version records for both classes and feature implementations, it would be possible to have version records associated with just classes. MViews aggregation (part-of) relationships between components automatically propagate a sub-component update record to its owning component and thus owning components are informed when their sub-components change. Update records associated with changes to a feature would thus be propagated to its parent class to be stored in the class' version record. While this approach means a smaller number of version records, it also means that local component version numbering can only extend to, for example, the class level, with no individual versions for feature implementations. The C-SPE view versioning scheme, described in the next section, also makes use of part-of relationships to reduce the number of version records.

## 4.3 View Versions

Views often express information difficult to express concisely in other ways, for example by the use of location to represent relationships between components not captured in the language semantics. It is thus useful to capture changes to views in a version system, in addition to the changes to base components. An important distinction between C-MViews and other environment models is its support for view versioning. C-MViews keeps view versions separate from base component versions, as a view may render several different base components. Changing any base component will thus partially change the view (and vice versa). Views may also change independently of their base components, as, for example, layout information is view-specific.

Fig. 3. shows view versioning for C-SPE. View components, such as class icons and connectors, do not have their own version records as they are not versioned independently from their owning view. A single base class or feature may appear in several different views (i.e. be linked to several different view class and feature components).

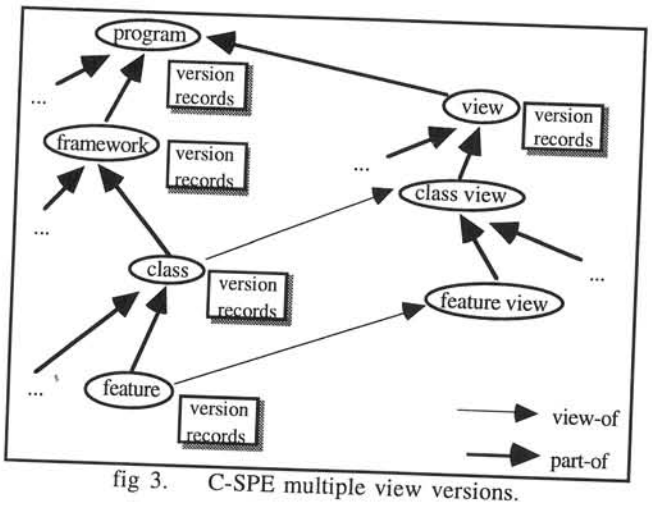


fig 3. C-SPE multiple view versions.

## 4.4 Regenerating Versions

The update records stored in a version record of a component (including update records specifying changes to the subcomponent configuration) can be used as deltas to regenerate previous or subsequent versions (by modifying the current version of a component). In order to convert one version to another in this way, C-MViews will undo the update records (to go back a version) or apply them (to go forward a version). Previous or subsequent versions may also be cached for efficiency.

## 4.5 Creating and Browsing Versions

Version records are stored separately from components in order to provide an always accessible version history. This information can be browsed using standard MViews tools.

Evolution graph views show several related component and view evolution graphs. These views are used to graphically specify new versions, alternate versions and alternate merges. They also allow developers to view the version record updates for a component, and to compare these with other components' version record updates.

## 5. Asynchronous Collaboration

Rather than the check-out style of SCCS, C-SPE permits an optimistic approach to version control in the parallel development of software by multiple developers. This approach is especially appropriate when the software under development cannot be easily partitioned between the developers. For example, the addition of a single function to an OO software system can lead to changes in several classes.

Asynchronous editing of multiple views (and, indirectly, software system components) is supported by supplying each developer with their own alternative versions of a view and the components rendered in the view. Changes to views (and thus to the software components rendered in the view) are recorded by their current versions. The change descriptions may subsequently be used to merge two (or more) alternatives or regenerate a particular version of a view or component. Alternative merging is carried out by a single developer with merge conflicts being detected and brought to the developer's attention. Our environment can animate updates on a view to illustrate, graphically or textually, changes other developers have made.

### 5.1 Asynchronous Editing

In C-SPE, developers have their own local repositories for component versions. A developer may create a new version of a component at any level in the component hierarchy, based on their current version. Once a new version is frozen it can be "exported"; ie, made available for others' to access. There is no problem with multiple copies of a version being distributed, as they are frozen before they can be exported. Another developer may later import an exported version and merge it with the developer's own version of the component, exporting the resulting version for other developers to use.

Fig. 4. illustrates this asynchronous editing and merging approach to collaborative development. Developer 1 copies version V1.0 of a component from developer 3 and creates a new version (denoted by V1.1a). Developer 2 also imports V1.0 and creates a new version (V1.1b) so they can modify it at the same time. After updating their alternatives, developers 1 and 2 freeze their component versions. Developer 2 then takes responsibility for integrating the changes, obtains a copy of version 1.1.a and merges it with version 1.1b. This merged component version is then exported as V1.2, for other developers to use.

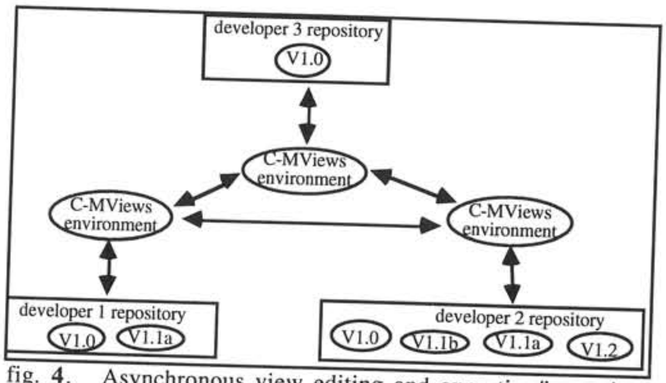


fig. 4. Asynchronous view editing and exporting/importing.

The component version control provided by C-SPE allows for changing and merging to be carried out at different levels within the hierarchy. This permits, for example, a developer to take a single component from another system version and merge just that with their current version. In this way, there need be no distinction between the versioning of libraries that are used across applications and the versioning of applications.

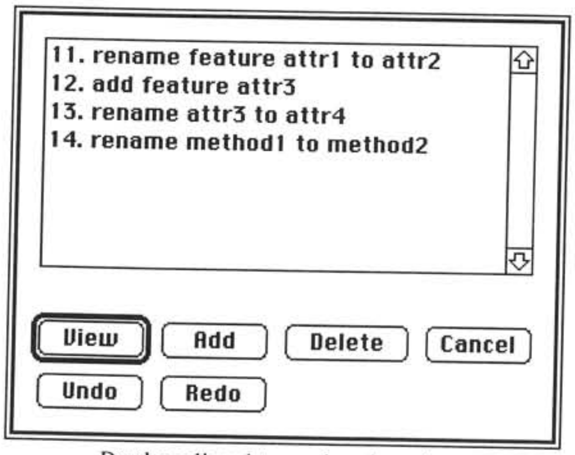
### 5.2 Asynchronous Merging

C-MViews expects specialisation environments, such as C-SPE, to provide facilities for capturing information about why a change is made, and to present this information appropriately in views. The capture and presentation of this information needs to be of limited "interference", as developers typically do not want to supply or see all of it every time they make a minor view modification. C-MViews attempts to overcome some of this interference by allowing an application to capture and present this information on demand via an update history dialogue used to browse version records. It is often difficult to decide when a developer wants to view the propagated information and currently this is done on demand via a dialogue.

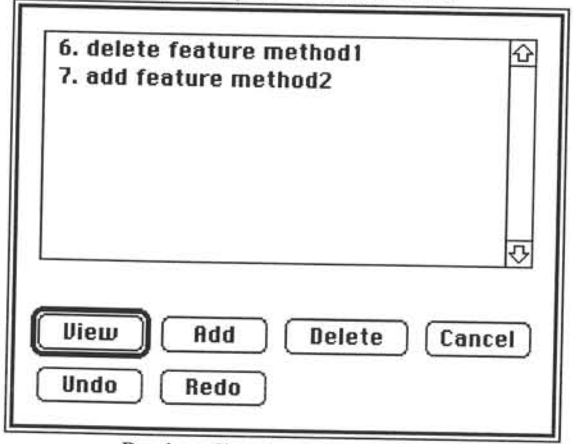
Alternatives may be merged by applying one alternative's update records to the other alternative's component, or by reverting to a common ancestor version and applying all update records from both alternatives to this earlier version. The first approach is faster, as there are likely to be fewer update records to apply. The second is more powerful (in general), as the ordering of the two streams of update records can be changed and thus conflicts are more easily resolved. Developers can have update records from another alternative incrementally applied to a view and watch how the view is updated (i.e. animate the affects of the merge). This allows developers to more clearly identify the effects of a merge operation in terms of actual changes to views.

Two alternative versions may contain conflicts that must be resolved when merging them. For example, one version deletes something that the other updates). C-MViews marks an update record if it can't be merged automatically and informs the developer of the conflict.

Consider the example in fig. 5. To merge the versions for a base class component, C-SPE must carry out both sets of update records. Note the conflict: developer 1 deleted feature method1 while developer 2 updated it. This problem should be identified and the developer merging the class alternatives informed of the conflict. The developer can then decide which update to allow (if either) or make other changes to reconcile the two alternatives. A similar problem occurs between the renaming of method1 by developer 1 and the addition of method2 by developer 2 (a semantic error). This can be resolved if method1 is deleted or if one of the updates is disallowed. C-SPE does not currently check for such semantic errors during alternative merging but identifies them when checking the semantic correctness of the merged class.


```
11. rename feature attr1 to attr2
12. add feature attr3
13. rename attr3 to attr4
14. rename method1 to method2
```

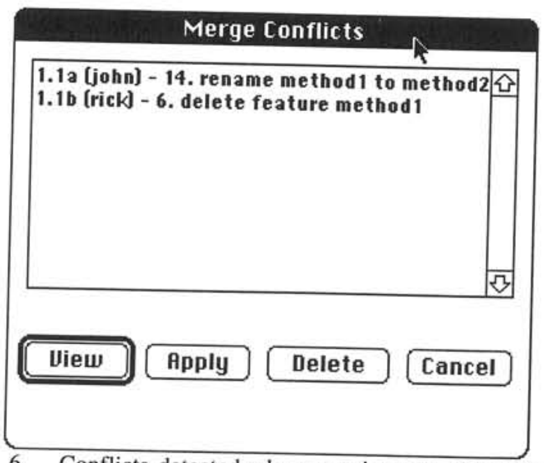
Developer 1's updates on class alternative


```
6. delete feature method1
7. add feature method2
```

Developer 2's updates on class alternative

fig 5. Example of two class alternative update lists to merge.

Fig 6 shows the Merge Conflicts dialogue which displays conflicts that C-SPE has detected in the merge process. The version record names are user-defined (although C-MViews stores unique version record identifiers internally).


Merge Conflicts

```
1.1a (john) - 14. rename method1 to method2
1.1b (rick) - 6. delete feature method1
```

fig. 6. Conflicts detected when merging two alternatives.

One interesting complication to the view merging process is MViews' support for free-edited textual views. These are stored as blocks of text which are parsed to recover structural information and thus to update base components shared by other views. C-MViews supports multiple versions of these "text forms", but only stores update records generated by the parsing process. Other aspects of the views which are updated between two versions of the same component (such as comments, expressions or code statements) need to be reconciled either manually or by using a traditional SCCS-style textual differencing approach.

## 5.3 Merging Multiple Views

Multiple views complicate the versioning process as a view has multiple versions and each base component rendered in the view has its own versions. Changing the current version of a view should modify underlying base components, if affected by the view change, and thus should also modify other views that render these base components. This can result in large scale changes from simple view version merging or when switching between a previous or subsequent view version. A further complication is that some view changes are view-specific, for example layout and view composition, eg which components and viewed and which aren't, while others affect the underlying base information, eg component renaming, adding or deleting relationships. Merging of two alternatives needs to resolve layout and composition conflicts (which often occur) with underlying base information conflicts (which occur less often). C-SPE must currently present the merging developer with a list of all conflicting update records for manual resolution. Heuristics may be useful to assist in automation of some of this decision making.

### 5.4 Indirect Merge Errors

At present, C-MViews requires that all components linked to a component whose version has been changed have their semantic values recalculated. This should be enhanced so that incremental semantic value recalculation is supported. C-MViews supports view update animation when merging view updates, but does not currently support a form of "active-diffing", as used by Mjølner environments. C-MViews does not indicate which information has been changed in alternate versions, but could do so via colouring graphical view icons and with techniques similar to those of Mjølner (+, - and underlining annotation) for textual views.

## 6. Synchronous Collaboration

The aim of synchronous collaboration is to allow two or more developers to simultaneously examine and alter a common view of a software system, communicating the changes they make between themselves as they are made. There are two main approaches to handling the integration of the changes made. The first approach is to consider that the developers are communicating and negotiating in order to derive a single result. In this case, it is appropriate that all the developers concerned share a common view, so that a change can be rejected by any participant and thus undone in all of their views.

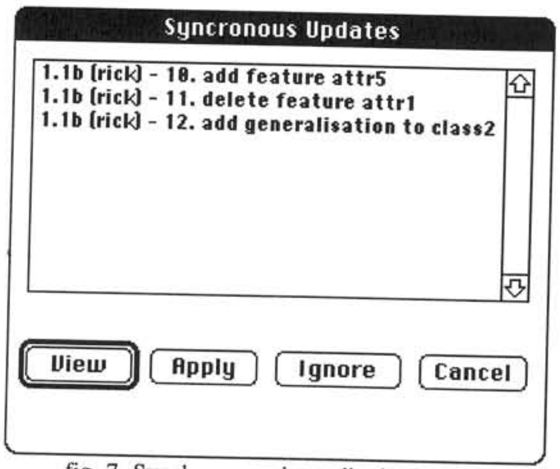
The second approach does not aim for a single result, so that developers may end up with their own distinct versions that reflect their current thinking. In this case, each developer can choose whether or not to accept the changes of others in the collaboration without affecting the others.

The framework that C-MViews provides can be used to support both of these approaches. It makes no assumptions about the choice of version that will form the basis of the collaboration and whether the whole system or specific (versions of) subcomponents will be used. A specific ISDE will determine how developers choose versions. For example a developer could request to be informed of changes to versions of a component (usually a view) that are descendants of a specified version of the component. Alternatively, the choice can be made on the basis of the developers concerned rather than the version. In the latter case, though, there are difficulties if the developers are starting from different versions, as the differences between them will need to be resolved before synchronous collaboration can begin.

Synchronous view editing in C-MViews is supported by broadcasting update records to other developers' environments as they are generated. These descriptions are then incrementally merged with other view alternatives, allowing collaborative development to take place. Developers may have these changes automatically applied to their views, may browse the descriptions before applying them, or may reject them, informing other developers why they have chosen to do so. Developers can move between asynchronous and synchronous modes of view editing; both modes are compatible under subsequent alternative merging.

Broadcast update messages include who, when and why information, which assists developers in understanding the change. User-defined update records can be broadcast to facilitate flexible, context-dependent, developer communication.

Fig. 7. illustrates the use of the C-MViews synchronous view editing model in C-SPE. The interface shown allows a developer to view update records describing the changes before deciding to apply them, ignore them, or view the affect of the changes on their version.


Syncronous Updates

```
1.1b (rick) - 10. add feature attr5
1.1b (rick) - 11. delete feature attr1
1.1b (rick) - 12. add generalisation to class2
```

fig. 7. Synchronous view edits in C-SPE.

C-MViews attaches a version record ID and unique sequence number to broadcast update records. Thus, no matter whether synchronous view editing is switched on or off for an alternative, changes the developer is not notified about can still be incrementally merged at a later date.

Update record broadcasting is handled as part of the general mechanism of propagating update records up the component hierarchy. The current version record for a component records other developers' interest in updates to the component. When an update record is stored in this version record, it is also broadcast to the environments of these other, interested developers.

These broadcast update records are then stored by the current version record of the other developer's component. Broadcast updates are presented to these developers when they work on the version record's component or view. An indication of new update records is given by shading icons or changing a menu bar item which results in a context-dependent communication mechanism. This is important in making C-MViews environments useable, so developers aren't inundated with messages at inappropriate times.

## 7. Implementation

The prototype of C-MViews is implemented in Snart and has been used to construct the prototype C-SPE environment. Here, we briefly comment on a few implementation issues and shortcomings in these prototype implementations.

### 7.1 Versions, Object Persistency, and Distribution

C-MViews extends persistent Snart object stores to support multiple versions of an object, so multiple component versions are represented by multiple object versions. Update records are now stored separately to the components that generate them. Update records include additional information over that used in MViews update records. This includes timestamp, user id, and change reason data. Each version record object contains a sequence of update records together with links to its predecessor(s) and successor(s), giving an evolution graph for the component.

C-MViews currently supports a common, shared component repository as a shared object store, and high-performance, single-user repositories. A database server is provided to moderate access to a shared object store. This allows a group of collaborating environments to provide high-speed data storage for each developer's alternatives, supports sharing of these alternatives, and handles update record broadcasting between developers. A central server is used, rather than developer-to-developer communication, so a definitive copy of the whole system is always available for new developers to access. This architecture also allows synchronous editing to be controlled from one location.

A component alternative in one developer's object store may be merged with a version in another's object store. Thus a component's object, its sub-component objects, and its version objects must be copied from one object store to another. As C-MViews knows about the aggregation structures present between software components, it can import and export the sub-components of a component automatically. For example, when a C-SPE class is imported, all of the features and relationships it owns are also imported. The shared repository acts as a form of distributed database by ensuring objects created in any developer's object space always have unique object IDs.

When editing different alternatives two developers may create different objects which represent the same conceptual view or base component. Subsequent merging of these alternatives will result in redundancy that can be resolved by any of the collaborators discarding one of these objects in favour of the other.

### 7.2 Limitations of the C-SPE prototype

Our current C-SPE prototype does not give developers any assistance in rearranging updates to resolve conflicts and at present only allows two versions to be merged at a time. Free-edited textual views may have multiple versions but C-SPE does not give any support to merging these alternatives (this must be done manually). C-SPE currently lacks a mechanism for relating different component versions. For example, if changes are made to several classes to implement one new system feature, these version relationships are not documented. It is thus difficult for developers to trace between related updates to different classes and frameworks. As a shared filing system is used to propagate update records, synchronous editing is quite slow and adversely affected by large network traffic volume.

## 8. Conclusions

Our experience in developing integrated software development environments indicates that both multiple views of software development and collaborative software development are important when building large software systems.

C-MViews is a new model for supporting multiple versions of software components and their multiple textual and graphical views. C-MViews supports collaborative development via multiple views, including both asynchronous and synchronous editing modes.

Developers can collaborate via asynchronous alternative editing and then merge alternatives. Changes between versions are recorded by sequences of update records, grouped into version records. Flexible version merging is supported using the update records, allowing developers to switch between component versions by undoing or reapplying these change descriptions, or by switching between multiple versions of the same component object.

Synchronous collaboration is supported by broadcasting update records as they occur between different developers' environments. These are incrementally merged into other versions of the same view, either automatically or manually. These modes of development are complementary and developers can switch between them.

Our experience with C-MViews indicates the need for several improvements to the architecture and its implementation. In particular, the user interface of C-MViews needs to be improved. For example, further work is needed to determine how changes broadcast for synchronous editing can be most usefully presented to developers. The mode of presentation is often dependent on the view being edited and individual developer's requirements. The capture of extra information, particularly a description of why a change was made, usually occurs above the update record level but below the version level. Thus a system for associating several related update records which implement one desired global change would be very useful. This could be extended so that updates to several different components can be related and traversed via hypergraph techniques, giving developers a high-level view of the relationships between different component versions. This is particularly useful for object-oriented systems, where changes are often made to several frameworks and classes to provide one new system function, but where individual updated components also need independent version control.

Use of semantic information during the version merging process may improve the simple heuristics used for detecting and managing merge conflicts. Support for tracing between changes made to analysis, design and implementation views in C-SPE should be provided. This is complicated by the versioning process as different versions of views and base components will be in use. Effective support for detecting and resolving semantic conflicts between views and components when changing between different versions is an open research issue. Another useful addition would be to allow developers to link different versions so changing from one component version to another also changes other component and view versions. It may also be useful for C-MViews to give ownership of components and views to some developers, to restrict component updating and alternative merging, if required by a development team.

## References

Ellis, C.A, Gibbs, S.J, Rein, G.L.: Groupware: Some Issues and Experiences, Communications of the ACM 34, 1 (January 1991).

Grundy, J.C., and Hosking, J.G., Constructing Multi-view Editing Environments Using MViews,Proceedings of the 1993 IEEE Symposium on Visual Languages, IEEE Press, August, 1993, pp. 220-224.

Grundy, J.C., Hosking, J.G., Fenwick, S., Mugridge, W.B., Connecting the pieces: integrated development of object-oriented systems using multiple views, Chapter 11 in Visual Object-oriented Programming, M. Burnett, A. Goldberg, T. Lewis Eds, Prentice-Hall, 1994 (in press).

Kaiser, G.E., Kaplan, S.M., Micallef, J.: Multiuser, Distributed Language-Based Environments, IEEE Software 4, 11 (November 1987), 58-67.

Magnusson, B, Asklund, U., Minör, S.: Fine-grained Revision Control for Collaborative Software Development", Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering, Los Angeles CA, 7-10 December 1993.

Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software 8, 1 (January 1991), 49-57.

Nascimento, C., and Dollimore, J. A model for co-operative object-oriented programming, IEE Software Engineering Journal 8, 1 (1993), 41-48.

OpenDoc, OpenDoc Technical Summary, OpenDoc Design Team, Apple Computer, Inc., 1993.

Ratcliffe, M., Wang, C., Gautier, R.J., Whittle B.R. Dora - a structure oriented environment generator, IEE Software Engineering Journal 7, 3 (1992), 184-190.

Reiss, S.P., PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering 11, (3) March 1985, 276-285.

Reiss, S.P. Working in the GARDEN Environment for Conceptual Programming, IEEE Software 4, 11 (November 1987), 16-26.

Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment, IEEE Software 7, 7 (July 1990), 57-66.

Reps, T., Teitelbaum, T. Language Processing in Program Editors, COMPUTER 20, 11 (November 1987), 29-40.

Roekind, M.J.: The Source Code Control System, IEEE Transactions on Software Engineering 1, 4 (December 1975), 364-370.