Support for Collaborative, Integrated Software Development

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

[*]Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand
[†]Department of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand
email: jgrundy@cs.waikato.ac.nz

A new model for supporting collaborative software development with shared multiple textual and graphical views is presented. Multiple views of software development can be synchronously, semi-synchronously and asynchronously edited by different developers. View versions can be incrementally merged, and view updates broadcast to other developers and incrementally incorporated as required in their alternative versions. The 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 programming environments which facilitate Computer Supported Cooperative Work (CSCW).

ISDEs support multiple tools, and multi-view editing in these environments allows developers to work with software components at different levels of abstraction, using different representations []. For example, 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. Consistency management is required to keep all of these views consistent under change.

Large software systems require the collaboration of multiple developers [,]. 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 []. Ideally environments should support both synchronous and asynchronous modes for all types of system data.

We describe a new model for constructing collaborative, multi-view ISDEs. Different environment tools and multiple views of the software under construction are integrated **by change description broadcasting via a shared data repository.** Asynchronous collaborative software development is facilitated **by sharing multiple** versions of software components between developers. Both fine-grained and coarse-grained versions of software components and multiple views are supported, with incremental version revision and merging. Synchronous and semi-synchronous **<<define this here?>>** collaboration are supported by real-time change broadcasting between environments, together with incremental change description presentation and version merging.

Section 2 discusses related collaborative ISDE research. Section 3 describes the user's perspective of our collaborative ISDE for object-oriented software development. Section 4 discusses the model this environment is based on and Section 5 describes the implementation of this model as a reusable object-oriented framework. Section 6 summarises the contributions of this research and outlines possible future research directions.

2. Related Collaborative ISDEs

SCCS [] supports version control for text source code files, with different versions regenerated and merged asynchronously. This technique does not work well for more structured information, such as diagrams, and the version merging process can be tedious and error-prone.

Mercury [] extends the Cornell Program Synthesizer [] to support a restricted form of collaborative programming. Changes to module interfaces are broadcast to other users, but program module versioning and multiple views are not supported. Nascimento and Dollimore [] describe a programming environment which supports manual, asynchronous version control for multiple programmers working on a shared Smalltalk program. Mjølner/ORM [] uses a fine-grained version control system to support both asynchronous and semi-synchronous editing. All of these environments provide **only one** view: a textual, structure-edited view of program code.

Garden [] supports software development via multiple textual and graphical views. Garden allows programmers to share software components and their views via an object-oriented database. FIELD environments [] allow Unix tools to be integrated by selective broadcasting of editing changes. Neither support true collaborative, multi-view editing.

Collaborative document editors support synchronous, collaborative work on a shared document []. They do not usually support asynchronous editing and version control, as the users are assumed to be working on the same document. ConversationBuilder [] provides flexible, active support for cooperative work activities, and is designed to facilitate tasks which can be constantly changing.

[] aim to support lazy consistency management for cooperative software development, where changes to software are broadcast before the changes are made. []

support multiple viewpoints for software development, with inconsistency management via logial predicates.

Dora [] provides multiple textual and graphical views of software developmen. It does not support the propagation of “partial” view updates between analysis, design and implementation views, and thus makes reconciliation of these views dependent on programmers remembering old updates.

3. C-SPE

SPE

SPE (Snart Programming Environment) is an ISDE which provides multiple textual and graphical views for constructing programs in Snart, an object-oriented Prolog []. SPE supports integrated analysis, design, implementation, debugging and documentation tools. Figure shows a screen dump from SPE with two graphical views (one for analysis and one for design), and two textual views (a class interface and a method implementation). There is full consistency management between all view types, so changes to one view are always reflected in other views that share the updated information, no matter how loose the connection between the view representations.

Additional analysis and design views, such as class contract and documentation views, as well as **graphical** and textual debugging views, are also provided by SPE. Graphical views are interactively edited and are kept consistent with other views by the environment directly updating changed icons. Descriptions of changes affecting graphical view components can also be viewed in dialog boxes.

Textual views are free-edited and parsed. Textual view consistency involves expanding descriptions of changes, called *change descriptions*, into the view’s text in a special header annotation. Some changes can be automatically applied by SPE to update the view’s text, such as renaming classes and features and adding or deleting features. Other changes represent “partial” changes affecting the view (eg a design level change propagated to an implementation view) which must be implemented manually by the software developer. 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.**<<How does it help determine consequences?>>**

Asynchronous Collaboration

C-SPE extends SPE to provide a collaborative object-oriented programming environment. SPE automatically generates descriptions of all view and software component updates as change descriptions. C-SPE uses these change descriptions to inform other programmers of changes applied at the analysis, design and implementation levels. This is done by broadcasting these change descriptions as they occur to other programmers’ environments and either storing them in different, shared versions or presenting them to programmers.

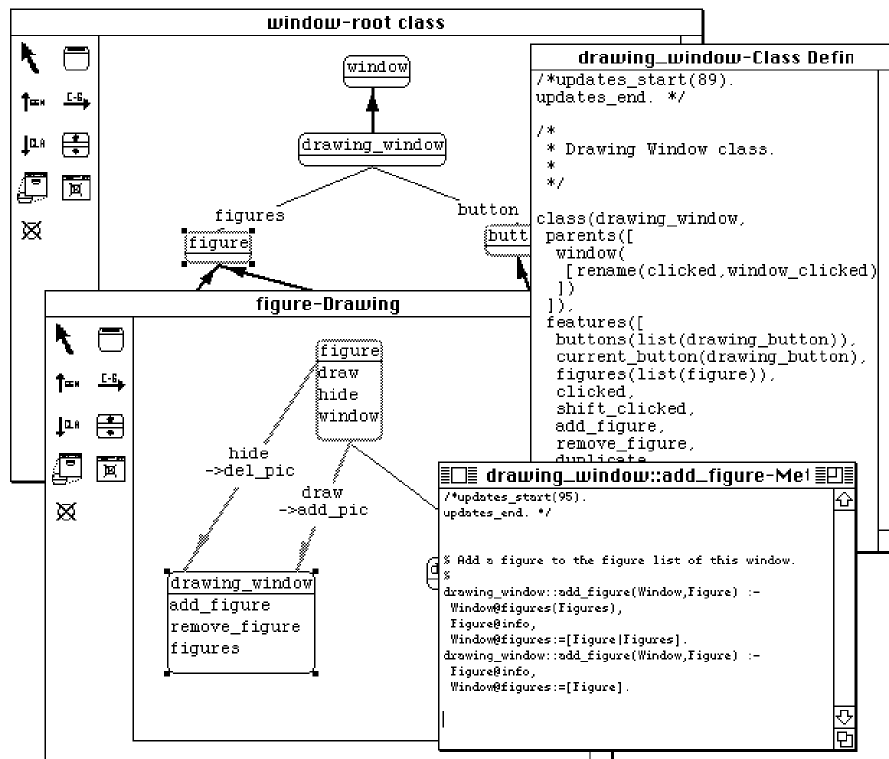


Figure . A screen dump from SPE.

The loosest form of collaborative software development supported by C-SPE involves sharing multiple versions of views, classes and class methods **among** software developers. After update by different developers, alternate versions of the same view or software component are merged to produce a new version.

Rather than the check-out style of SCCS, C-SPE **adopts** an optimistic approach to version control**. This** 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.

"Evolution graph" **<<Diagram of this type of graph or ref>>**views s**how the relationships between component and view versions. They** are used to graphically specify new versions, alternate versions and alternate merges, and allow developers to view all change descriptions in different versions (the "modification history" of a view or software component).

Developers edit their own versions of views, and/or the software components rendered in the views asynchronously. Developers may create and modify new versions of a component based on their current version. This "freezes" the current version (ie new changes are locked out), and allows it to be exported for other developers to use. Another developer may subsequently import an exported (frozen) version, and merge it with his/her own version, exporting or further modifying the resulting version.

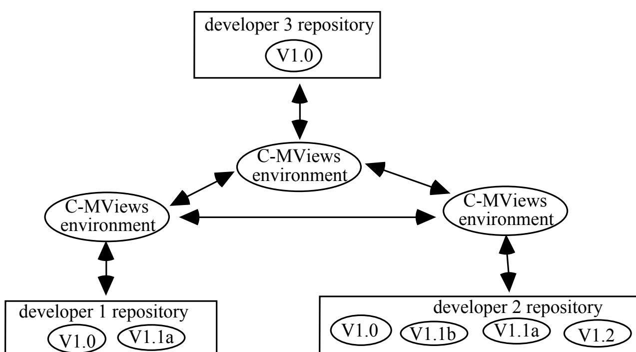
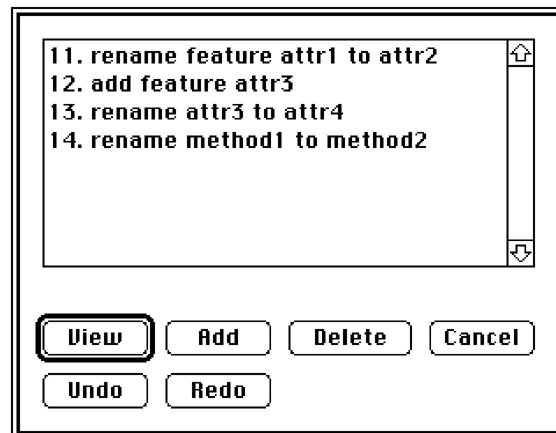


Figure . Asynchronous view editing and exporting/importing.

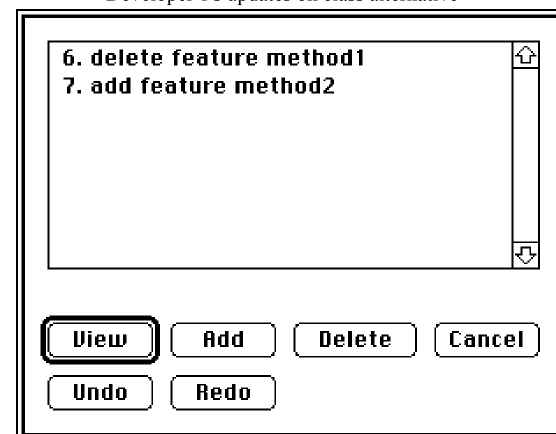
Figure illustrates this asynchronous editing and merging approach to collaborative development. Developer 1 copies version V1.0 of a component from developer 3 and creates a new version (denoted by V1.1a). Developer 2 also imports V1.0 and creates a new version (V1.1b), so they can modify it at the same 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.

Alternatives are merged by having C-SPE apply change descriptions selected from one alternative version to the other alternative's view or component, or reverting to a common ancestor version and applying all change descriptions from both alternatives to this earlier version. Developers can choose a subset of the version's change descriptions to have C-SPE apply, and can have groups of change descriptions applied out of the sequence they occured, if desired.

C-SPE applies the change descriptions to be merged incrementally to a view. Developers can observe how the view is updated by each incremental change, thus animating the effects of the merge. This allows developers to more clearly identify the effects of a merge operation in terms of actual changes to views.



Developer 1's updates on class alternative



Developer 2's updates on class alternative

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

Two alternative versions may contain conflicts that must be resolved when merging them. For example, one version **may delete** something that the other updates. C-SPE identifies change descriptions it can't **merge** automatically and informs the developer of the conflict. For example, in Figure C-SPE must carry out both sets

of change descriptions, but developer 1 **has** deleted feature “method1” while developer 2 **has** updated it. This problem is identified by C-SPE and the merging developer 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 **with** 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. Figure shows the Merge Conflicts dialogue which displays conflicts that C-SPE has detected.

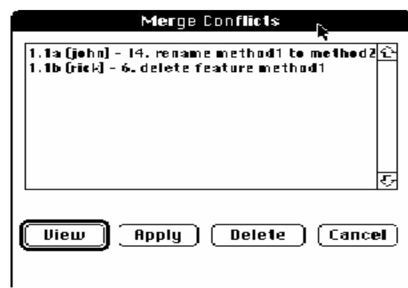


Figure . Conflicts detected when merging two alternatives.

One interesting complication to the view merging process is SPE’s support for free-edited textual views. These are stored as blocks of text which are parsed to recover structural information and thus to update information shared by other views. C-SPE supports multiple versions of these text components, but only stores change descriptions 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. **<<this para could go>>**

Synchronous Collaboration

Synchronous collaboration allows two or more developers to simultaneously examine and alter a view, communicating the changes they make between themselves as they occur. 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 **the shared** 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. This results** in “semi-synchronous” collaborative editing.

C-SPE supports both types of collaboration. With semi-synchronous collaboration, developers have their own alternative and are incrementally informed of updates other developers are making to their alternative versions by the receipt of change descriptions, which are then displayed in dialogs or textual views. With synchronous collaboration, developers share the same version and view updates are shown simultaneously in other developers’ views.

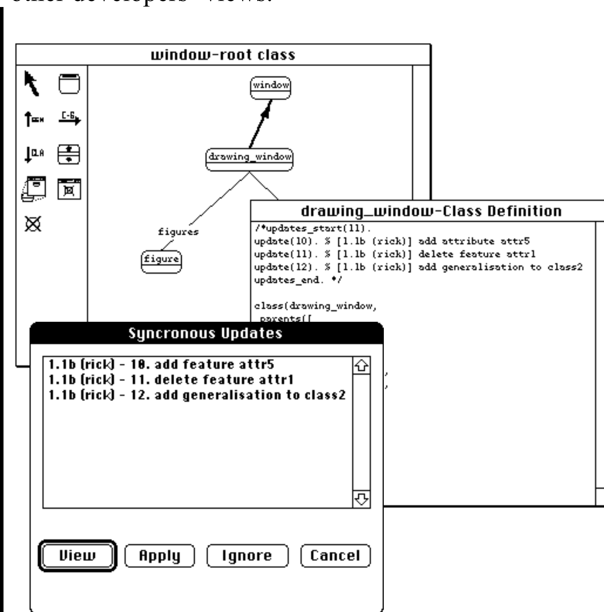


Figure . **Semi-synchronous** view edits in C-SPE.

Figure illustrates semi-synchronous view editing in C-SPE. The dialog shown allows a developer to view change descriptions **of** the changes before deciding to apply them, ignore the changes, or view the effect of (some or all of) the changes on their version. Developers can request C-SPE to automatically merge received change descriptions with their current alternative incrementally as they arrive. Change descriptions can also be viewed in textual view headers.

During synchronous collaboration, developers share the same version of a view. Changes made by one developer are reflected immediately in the view the other developers interact with. Whereas view alternatives edited semi-synchronously may have different layouts and viewed components, synchronously edited views are always identical among all collaborating developers.

Developers can move between semi-synchronous and asynchronous development at will. To move from synchronous to asynchronous collaboration, however, developers must obtain the shared version, and create an alternative of it, before asynchronously editing it.

4. C-MViews

We now describe C-MViews, the framework used to construct C-SPE, commencing with a description of the

single user MViews framework, and following with extensions to support collaborative environments.

MViews

SPE is implemented **as a collection** of Snart classes, specialised from the MViews framework []. MViews supports the construction of new ISDEs by providing a general model for defining software system data structures and tool views, with a flexible mechanism for propagating changes between software components, views and distinct software development tools.

As shown in Figure , ISDE data is described as components with attributes, linked by a variety of *relationships*. Multiple views are supported by representing each view as a graph linked to the base software system graph structure. Each view is rendered and edited in either a graphical or textual form. Distinct environment tools can be interfaced at the view level (as editors), via external view translators, or multiple base views maybe connected via inter-view relationships, as described in [].

When a software or view component is updated, a change description is generated. This is of the form UpdateKind(UpdatedComp, UpdateKind-specific Values). For example, an attribute update on Comp1 of attribute Name might be represented as: update(Comp1,Name,OldValue,NewValue)

All basic graph editing operations generate change descriptions and pass them to the propagation system. Change descriptions are propagated to all related components dependent upon the updated component's state. Dependents interpret **these update descriptions** and possibly modify their own state, producing further change descriptions. The change description mechanism **supports** a diverse range of software development environment facilities, including semantic attribute 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 [].

New software components and editing tools are constructed by reusing abstractions provided by an object-oriented framework. ISDE developers specialise MViews classes to define software components, views and editing tools to produce the new environment. A persistent object store is used to store component and view data.

Multiple Versions

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

Several approaches to managing hierarchical versioning exist: a new version of the whole system can be created whenever any change is made; individual version numbers at a particular level of the component hierarchy can be maintained; or individual versions for any component in the hierarchy can be stored, with a configuration management tool used to reconstruct a system version from lower-level sub-component versions.

Figure The MViews Architecture.

C-MViews aims to support all of these approaches by providing a tailorable low-level versioning mechanism, based on stored sequences of change descriptions, called *version records*. Version records can be associated with any C-MViews component or view, and contain a record of changes made to that component (as change descriptions) since the previous version. These can include change descriptions describing changes to the component itself, its sub-components, or the configuration (version used) of sub-components.

For example, Figure shows how C-MViews version records are used for C-SPE's version control. C-SPE adopts a component-level versioning approach where each component in the component hierarchy **down to the level of individual classes** has multiple versions with associated version records.

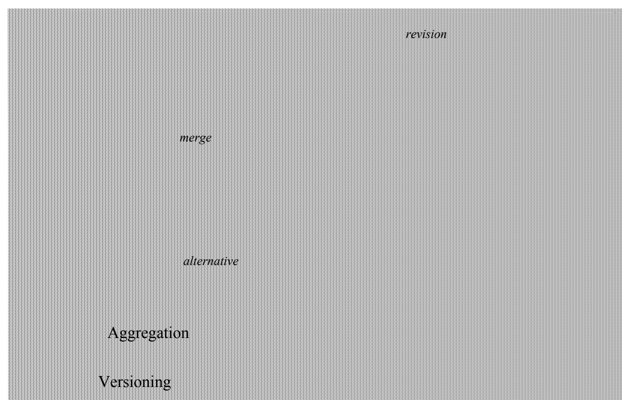


Figure .	The general structure of C-SPE versions. **<<The version record should be removed from the feature in the diagram>>**

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. When a new C-SPE component version is created a new version record for that component is also created. The component's parent, eg the feature's class, is also notified of this new sub-component version. If the parent's version is **frozen,** a new version of the parent is **also** created.

For small sub-components it is useful to record changes only in their parents' version records, for efficiency. For example, in C-SPE, version records are associated with classes but not individual class features. This reduces the number of version records needed, but means versioning only proceeds down to the class level, with no individual feature versions. C-MViews aggregation (part-of) relationships between components automatically propagate a sub-component (eg class feature) change description to its parent component (eg class) which can then store the change description in its version record.

View Versioning

An important distinction between C-MViews and other ISDE models is its support for view versioning. View versions are kept 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. Figure shows view versioning as used by C-SPE. In this case a single version record is held for the view as a whole rather than having individual records for view components, such as class icons and connectors.

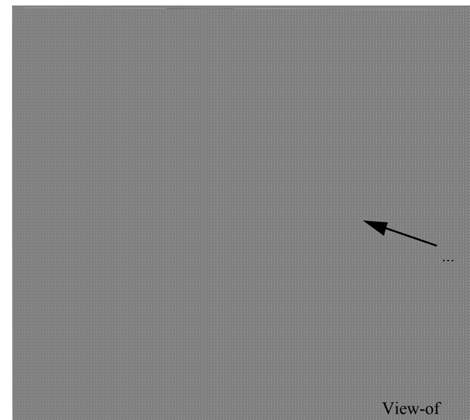


Figure .	C-SPE multiple view versions.

Views complicate the versioning process, as each view can have multiple versions and each base software component rendered in the view can also have multiple versions. Changing the current version of a view will modify affected base components, which should also cause modifications to 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-MViews currently presents **the developer** with a list of all conflicting change descriptions for manual resolution. Heuristics may be useful to assist in automation of some of this decision making.

Version Merging

Change descriptions stored in a version record, including change descriptions specifying changes to sub-component configurations, are used as deltas to **(re)**generate previous or subsequent versions. In order to convert one version to another, C-MViews will undo the change descriptions (to go back a version) or apply them (to go forward a version). Previous or subsequent

versions may also be cached for more efficient configuration management.

C-MViews **based** environments, such as C-SPE, provide user interface 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 the evolution graph dialogs used to browse version records.

During version merging, any structural update conflicts encountered during merging (such as deleting components updated in another version) are identified and presented to the merging developer. Semantic errors can be detected by incrementally reevaluating semantic attribute values and constraints after each change description merge. Any "error" change descriptions generated can be presented to the programmer, as they indicate a semantic merge conflict has occured.

View update animation during merging is achieved by updating and re-rendering view components for each change description merged. Developers can choose a sequence of change descriptions to merge with another version, and can step through the application of each change as it is applied, seeing the view dynamically updated. Unlike most other systems, developers can even merge version updates out-of-sequence, with C-MViews detecting any structural or semantic conflicts this produces.

Synchronous Collaboration

Synchronous and semi-synchronous collaboration are supported by broadcasting change descriptions to other developers' environments as they are generated, as shown in Figure .

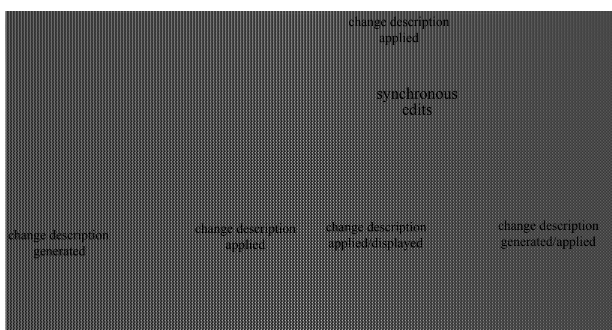


Figure . Synchronous software development.

To support semi-synchronous editing, broadcast change descriptions are received by other developers and cached in special version records. They are then presented in dialogs or textual view headers to inform other developers of changes made to other alternatives. These change descriptions can then be incrementally merged with other developers' view alternatives using the same merging techniques as employed for asynchronous merging.

To support synchronous editing, no developer "owns" the shared version of a view. All updates attempted by developers are sent to a central server which then updates the shared version itself. Generated change description(s) are then broadcast to all collaborating developers' environments, whose views are then re-rendered to reflect the change. Fine-grained locking is maintained by the server so only one edit of the same view component is accepted at a time.

Change Description Broadcasting

Change description broadcasting is handled as part of the general C-MViews mechanism of propagating change descriptions. The current version record for a component records other developers' interest in updates to the component. When **a** change description is stored in this version record, it is also broadcast to the environments of these other, interested developers. Shared version records used for synchronous editing also co-ordinate updates made to the shared version.

Broadcast change descriptions are either stored by the current version record of the other developer's component or applied immediately to shared versions. Stored updates are presented to these developers when they work on the version record's component or view. An indication of new change descriptions is usually 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.

Broadcast change descriptions include who, when and **optional** why information, which assists developers in understanding why the changes have been made. User-defined change descriptions can also be broadcast to facilitate flexible, context-dependent communication.

C-MViews also timestamps each broadcast change description by attaching a version record ID and unique sequence number. They are then stored in special "broadcasted" and "received" version records in each developers' environment. Thus, no matter whether semi-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, using the timestamp information.

This also helps to support fault-tolerance, **eg** when one developer's network connection or machine fails. Change descriptions cached by the developer's environment and the central server can be rebroadcast when the connection is re-established. Failure of the central server prevents any form of synchronous collaboration form taking place, but as developers have their own alternatives, asynchronous development can continue. We plan to extend C-MViews to support synchronous collaboration between developers without using the central server, which will improve the robustness of resulting environments.

5. Implementation

A prototype of C-MViews has been implemented in Snart and has been used to construct the prototype C-SPE environment. C-MViews extends persistent Snart object stores to support multiple versions of an object, for multiple component versions. Change descriptions are stored in these version records, and include additional information, such as time-stamp, user id, and change reason. Each version record object contains a sequence of change descriptions together with links to its predecessor(s) and successor(s), giving an evolution graph for the component.

C-MViews currently uses 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 the 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 change description 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. This 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.

The shared repository acts as a form of distributed database by ensuring objects created in any developer's object space are always unique. 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.

Our current C-SPE prototype detects structural and semantic conflicts as they occur during the merging process, and presents invalid or error change descriptions to the merging developer. C-MViews currently does not, however, 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.

6. Conclusions

Our experience in developing integrated software development environments indicates that multiple views of software development, integrated development tools, and collaborative software development are important when building large software systems. C-MViews supports multiple versions of software components and their textual and graphical views. Tools are **integrated** at the view level or data repository level. Collaborative development is via asynchronous, semi-synchronous and synchronous editing of multiple views. Semi-automatic version merging is supported, including incremental version merging. Two forms of synchronous collaboration are supported by broadcasting change descriptions between developers' environments.

Experience with C-SPE suggests that asynchronous development is most useful for low-level design and implementation **views**, or when major system changes are being carried out. Semi-synchronous development is useful for higher-level collaboration where different alternatives are maintained by each developer. Synchronous development seems most appropriate when high-level designs are being worked on and alternatives are not desired during the collaboration process.

We are currently working on several improvements to C-MViews and C-SPE. This includes determining how changes broadcast via synchronous and semi-synchronous editing can be most usefully presented to developers. The capture of extra information, particularly a description of why a change was made, usually occurs above the change description level but below the version level. We are experimenting with various "development tasks" for each developer and for groups of developers. These are used for associating groups of related change descriptions and for relating change descriptions on different components, 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 classes to provide one new system function. Propagation of partial changes between higher-level software components, as done for multiple view consistency, would also assist in supporting programming-in-the-large.

Acknowledgements

The helpful comments of the anonymous reviewers on the draft of this paper are gratefully acknowledged.

References