Keeping Free-edited Textual and Graphical Views of Information Consistent

John C. Grundy

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

John G. Hosking

Department of Computer Science
University of Auckland
Private Bag, Auckland, New Zealand
john@cs.aukuni.ac.nz

Abstract

Multi-view editing is useful in many situations where users of a software application want to see and interact with different representations of the same information. This paper describes a new approach to keeping free-edited multiple textual and graphical views of information consistent. Descriptions of changes to information items are displayed in various ways in the multiple views of these items. Users can request an editing tool to automatically apply changes to a view, select a change to make from a range of possible changes, or manually implement changes to maintain view consistency. Semantic errors, user-defined changes and hierarchical changes can be represented, and this technique also supports flexible view consistency for cooperative work systems. Experience with this technique in several diverse multi-view editing environments is described.

1. Introduction

Multi-view editing provides software application users with multiple, editable views (perspectives) of a common information model, as shown in figure 1 [4, 29]. This allows users to view and interact with the information in different ways. Some representations are better suited than others for the various tasks involved in software system specification, design and implementation.

One consequence of a shared information model is that users of multiple views usually want each view to be consistent with the other views [4, 29, 33]. When one view is updated, the other views which share the affected data must be updated to reflect the change. This involves propagating the effect of a view change to the common information model and then to all other affected views (i.e. view consistency management). Sometimes it is desirable to maintain view inconsistency, for short or long periods of time [8, 30, 20], and consistency management is further complicated when multiple users cooperate to edit multiple views of shared information [7, 31].

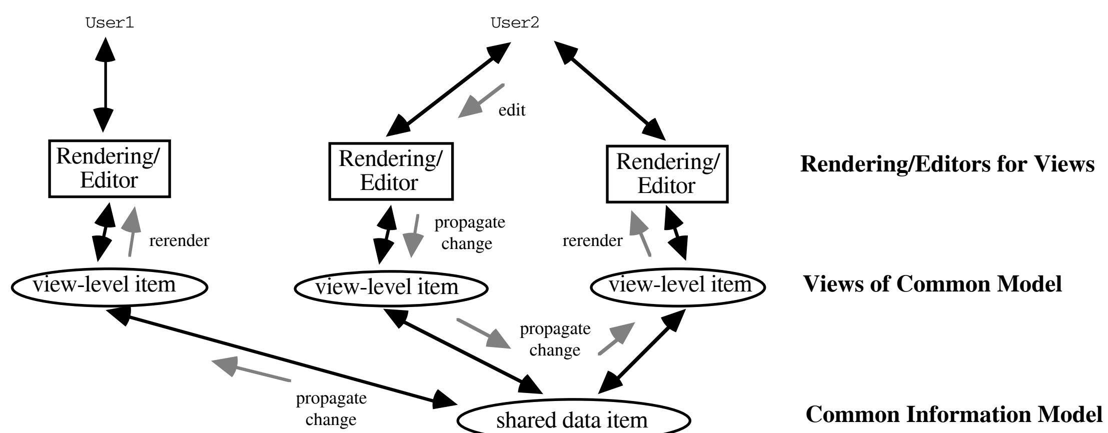

Figure 1. The concept of multi-view editing.

Many systems employ multi-view consistency, including CASE tools, programming environments, Integrated Software Development Environments (ISDEs), interface builders, and CSCW applications. Tools for building such multi-view editing applications include user interface toolkits, editor toolkits, program visualisation systems, environment generators and database systems. However most existing multi-view editing systems support limited forms of consistency management, with only direct translations

of editing changes supported between views. Many systems also require views to be structure-edited or employ ad-hoc and/or difficult to reuse techniques. This severely limits the degree of true inter-view consistency (and inconsistency) management that can be achieved.

This paper describes a new approach for supporting consistency management in multi-view editing environments. Multiple views are either interactively edited (graphical views) or free-edited (textual views), for maximum flexibility and suitability to user editing preferences [2, 43]. The kernel idea in our approach is to make view inconsistencies visible and interactible for environment users. Inconsistencies between views caused by a user editing a view are presented in affected views as descriptions of changes in dialogs and headers, view annotations and colouring, and option choices. Users view and interact with these representations of inconsistencies by asking the environment to automatically make a view update, manually making an update, or selecting a desired update option. Inconsistencies can be hierarchically grouped, ignored and deleted, or retained in modification history lists, and can be used to support flexible versioning and cooperative work facilities.

Section 2 reviews existing multi-view consistency management applications and models. Section 3 describes how our approach presents inconsistencies to users. Section 4 discusses interaction techniques with these inconsistency representations. Section 5 briefly presents the design and implementation of an architecture for building new multi-view editing environments utilising these techniques. Section 6 illustrates the versatility of our technique and architecture in relation to various applications we have built with it. Section 7 summarises the contributions of this research.

2. Existing Multi-view Editing Systems

Figure 2 shows a screen dump from SPE (Snart Programming Environment), an ISDE for developing object-oriented programs using Snart, an object-oriented Prolog [11]. SPE supports multiple textual and graphical views of software development, including full bi-directional consistency management between all views.

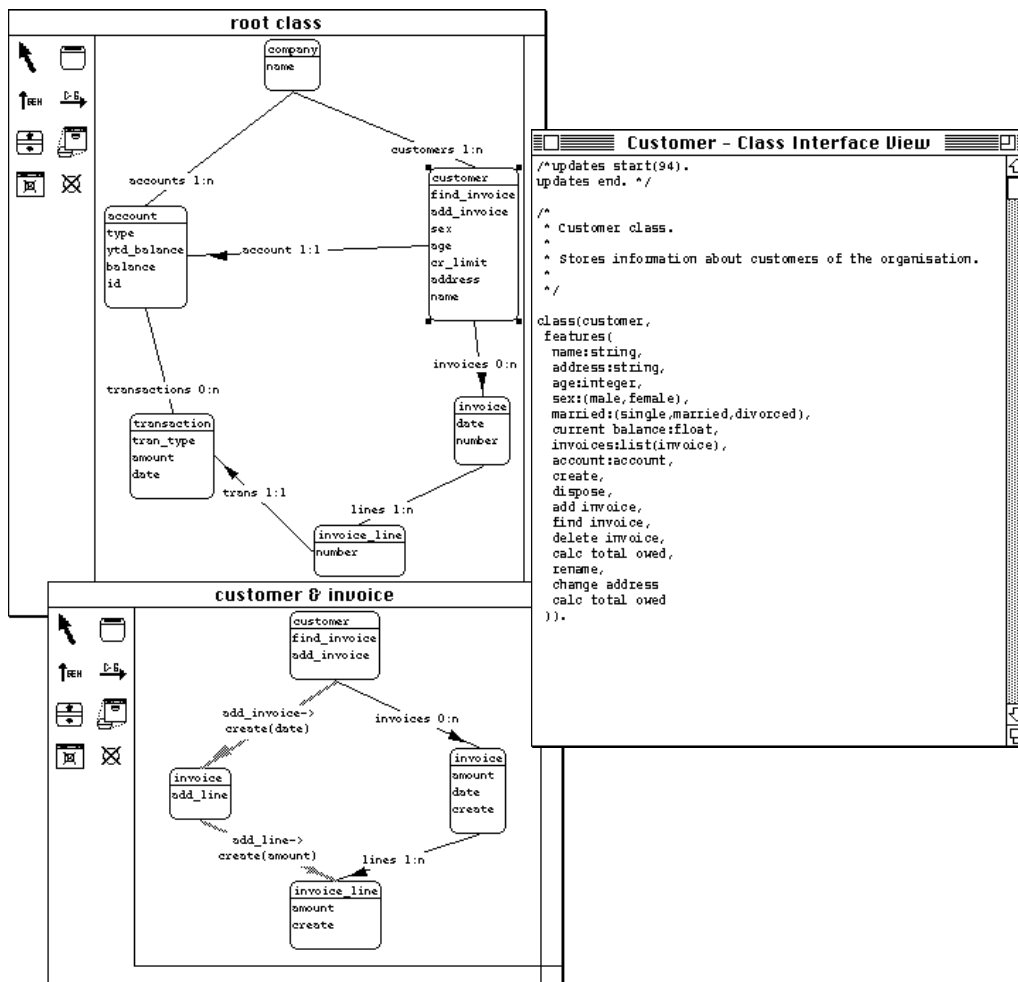

Figure 2. An example multi-view editing environment.

We use SPE to illustrate a number of consistency management issues not well supported by current multi-view editing environments and models. Later, we show how our model can be used to provide consistency and inconsistency management for SPE and other ISDEs built using our model. One consistency requirement for SPE is that textual views need to be kept consistent with graphical views, and vice-versa. Many changes to graphical views, however, cannot be directly translated into textual view updates. An example is when a client-supplier (method call) relationship is added in a graphical design view. This cannot be automatically translated into a textual view update, as neither the call's arguments nor its position in the textual code are known. Similar problems occur when trying to keep graphical analysis and design views, textual class interface and method implementation, and graphical and textual documentation views consistent [11]. In addition to managing consistency between views supporting different phases of software development, SPE also needs to support consistency between multiple views of a single program description, with views sharing a common information model. Collaborative software development must also be supported, with multiple views being shared and kept consistent for multiple developers.

Many systems, such as those developed on top of Unix, provide no consistency management between multiple tools and views. Changing information in one view is not reflected in other views which share the information, and different tools are usually implemented as separate, disconnected entities [23, 29]. FIELD environments [35, 36] and succesors, such as DECFUSE [19], as an exception, use selective broadcasting to propagate information changes between multiple Unix tools which share a consistent user interface look-and-feel. Only limited forms of view consistency are supported by FIELD, however, and building such environments and integrating new tools into the environment requires much effort [29].

Visual programming environments, such as Prograph [5] and Garden [34], user interface builders, such as Unidraw [41], and CASE tools, such as Software thru Pictures™ [42] and TurboCASE™ [39], provide multiple views of programs and designs. Views are usually graphical, with only changes that can be automatically applied by the environments being propagated. Textual view "consistency" is usually limited to regenerating textual code, and reverse-engineering is used to partially update design views when text is modified. This results in disjointed environments for users, with many changes having to be manually propagated between views.

FormsVBT [3], implemented using the Zeus program visualisation system [4], provides integrated textual and graphical views for user interface specification. A free-edited textual view is kept consistent with an interactively edited graphical view. FormsVBT, however, requires graphical view updates to be locked out when the textual view is being updated, greatly constraining a user's choice of editing mode. Ad hoc techniques are used to keep the simple S-expression textual code consistent, which do not scale up to more complex infomation representations.

Many ISDEs provide multiple views of software development, including Dora [32], PECAN [33], and MELD [21]. These environments utilise restrictive structure-editing to keep their views consistent [2, 43], and only support the propagation of changes which can be automatically applied to views by the environment.

A few systems provide inconsistency management support, where multiple views need to be inconsistent for some time, but this inconsistency needs to be recorded and resolved at a later date. An example is [8], which uses logic predicates to record inconsistencies between different viewpoints on software designs. However, this approach utilises complex, cumbersome logic expressions to express the inconsistencies between multiple views.

Many environment generators, models and software architectures have been developed for building new applications supporting multi-view editing. Examples include the Smalltalk MVC framework [24], Interviews [25], the ItemList structure [6], and Zeus [4]. However, most of these systems provide no support for the kinds of flexible (in)consistency management required in environments like SPE, and are very difficult or impossible to extend to support such techniques.

3. Presentation of Inconsistencies

Any environment supporting multiple views should keep these views consistent under change, or at least inform users when a view is presenting inconsistent information [29]. It is useful in the latter case to have the environment describe the inconsistent nature of the view information. This can be by describing the update made to another view making this view inconsistent, presenting a range of operations on the view that need to be made to resolve the inconsistency, or by indicating the view is inconsistent and providing

more detailed information elsewhere. Our approach utilises all of these techniques to present users with representations of view inconsistencies. As inconsistencies caused by multi-view editing often can not be automatically resolved by the environment, users need to be informed of view inconsistencies via presentation techniques, and then be allowed to interact with these representations to resolve inconsistencies.

We have built an architecture which automatically generates descriptions of changes made to view items, called *change descriptions*, whenever view items are updated. These are propagated to other views which share the updated information, and these views can respond to the change descriptions by updating their own state, presenting the change descriptions to users, or ignoring the change to the updated view. Presentation of change descriptions allows users to visualise inconsistencies between views. This architecture is described in more detail in Section 5. This section illustrates different ways of presenting view inconsistencies by using change descriptions. The example environments used have been built using our architecture, and all of the techniques described are general and not specific to each system.

3.1. Automatic Update of Views

Some view updates are easy for environments to automatically carry out. For example, in SPE renaming a class icon is easy to propagate and automatically implement in all other graphical views in which the class is shown. All graphical icon objects for the class are updated and rerendered to reflect the class rename. All of the existing multi-view editing models described previously support this simple change propagation mechanism. Most, however, only support this level of view consistency - if the update can be directly made to another affected view, then it is performed by the environment. If the update cannot be made, it is not performed and the user of the environment is usually never informed of the possible view inconsistency. In addition, many environments do not support translation of change between views supporting different notations, when this could be automatically made. For example, many CASE tools do not support change propagation between OOA and ER or NIAM views [40].

3.2. Presenting Descriptions of Inconsistencies

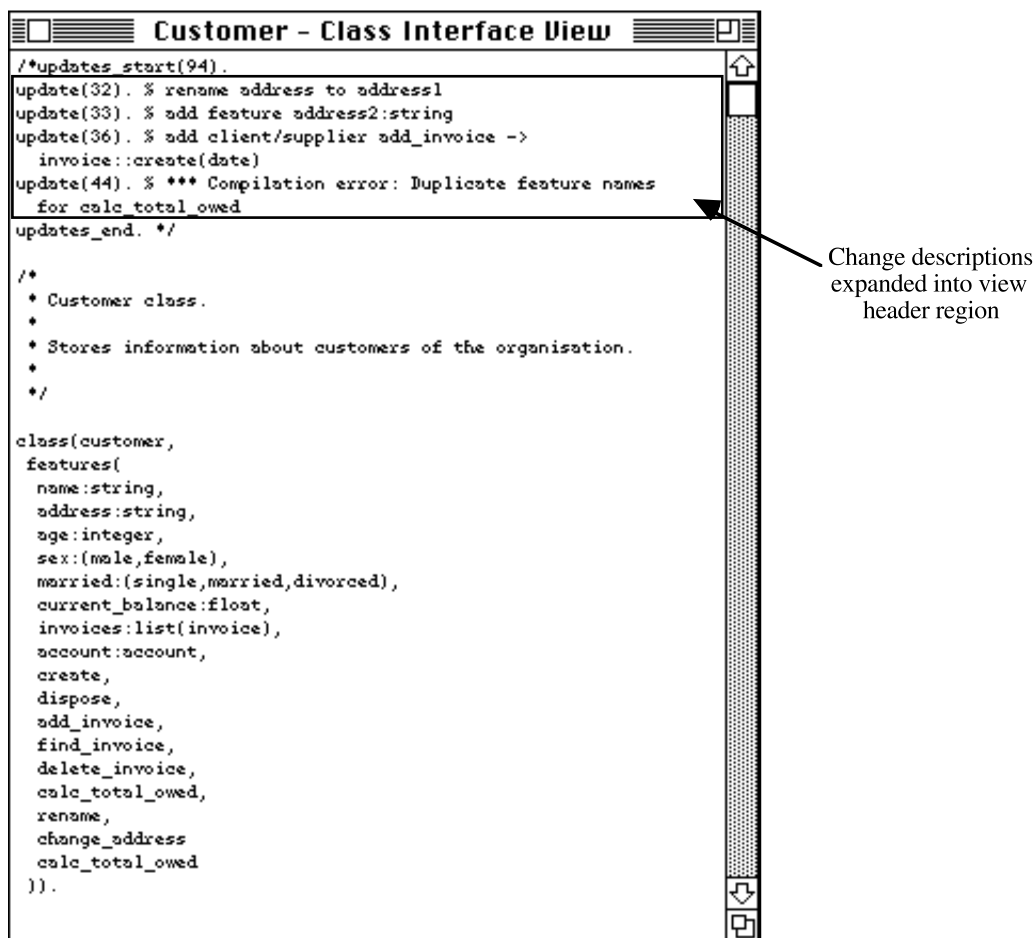

Figure 3. Indicating inconsistencies in textual views.

Some view updates can not be directly translated into appropriate updates on other views. This often occurs when the updated view and other affected views represent the shared information in quite different ways, or at different levels of abstraction. For example, SPE textual views show class interface definitions and method implementation code, which are at a lower level of abstraction than graphical analysis and design views. Keeping these views consistent, or at least informing programmers of inconsistencies between the views, is more difficult than keeping the design views consistent with each other.

Our multi-view editing approach solves this problem by presenting users with a description of any inter-view inconsistency that may be present. This is done by inserting *change descriptions* into the view. For example, figure 3 shows an SPE class interface view with several change descriptions expanded into a header region at the start of the view's text. These describe inconsistencies between this view and other, modified views of the program. These change descriptions can be used to present inconsistencies between any kinds of views.

In this example, change #32 indicates that the programmer has renamed the address attribute to address1, change #33 indicates a new attribute address2 has been added, change #36 indicates a client/supplier relationship between the customer::add_invoice method and the invoice::create method has been added, and change #44 indicates that a compilation (semantic) error has occurred, with duplicate calc_total_owed methods being defined for the customer class. The first three of these changes might have been made in a graphical view and thus the change descriptions are informing the programmer of (possible) view inconsistencies between this textual view and the modified graphical view.

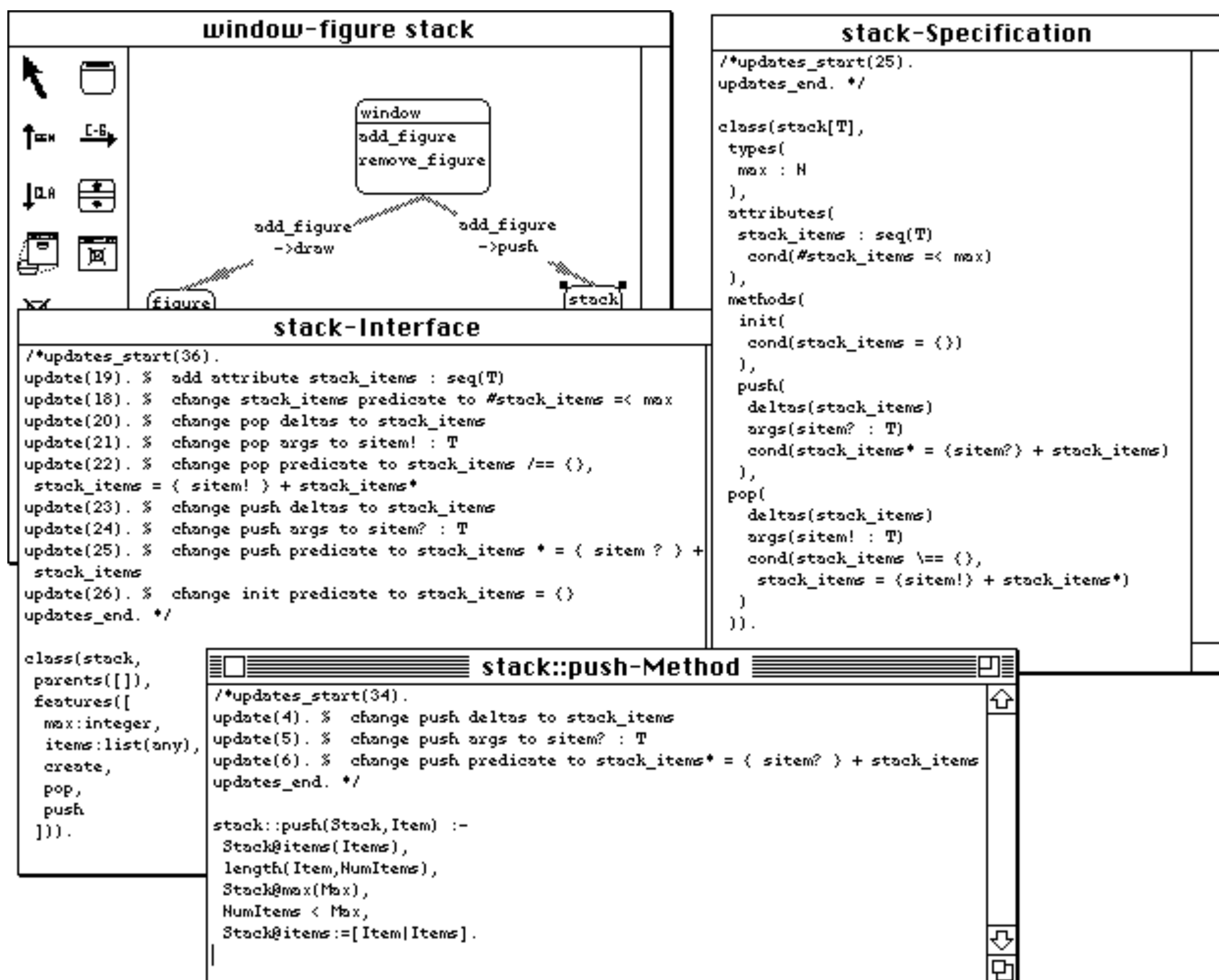

Figure 4. Keeping textual views consistent with one another.

Changes 32 and 33 can be automatically applied by the environment to the textual view to update it, and, in SPE, the programmer can specify that the environment is to always automatically update the view's text to reflect such changes, rather than displaying the change descriptions. Changes 36 and 44 can not be automatically carried out by the environment. To resolve change 36, for example, the programmer must, at some later time, modify the textual view of the customer::add_invoice method to insert an appropriate

method call with appropriate arguments, and possibly update the customer class textual view so that an appropriate reference to invoice class objects exists.

Changes from graphical view to graphical view, and from textual view to textual view, can also be presented in this way. For example, figure 4 shows an Object-Z-like view in SPE being kept consistent with a method implementation view. Both views are textual, and changes to one type of view are shown in the other type by change description presentation. This approach is also used to keep graphical analysis and design views consistent, with change descriptions shown in dialogs.

There are several other ways to present such change descriptions in views. Header regions in different parts of a view and hypertext buttons allow users to access groups of change descriptions in either the view itself or in pop-up menus or dialogs. We have found scrolling dialog menus to be most appropriate when a potentially large number of inconsistencies are, or may be, present. Pop-up menus work well when a smaller number of inconsistencies need documenting. View annotation, as used for the header section in SPE textual views, is useful when users want or need to see all relevant view inconsistencies at the same time.

3.3. Semantic and User-defined Inconsistencies

Change descriptions can present semantic and user-defined inconsistencies between views, in addition to the structural inconsistencies described above. Change 44 shown in figure 3 describes a semantic error which has occured as a result of editing a view. SPE provides an option to locate classes whose definition has semantic errors. These change descriptions can be viewed for each class, and views containing representations of each class can be highlighted to indicate the presence of inconsistencies (via graphical icon shading/colouring and textual name highlighting).

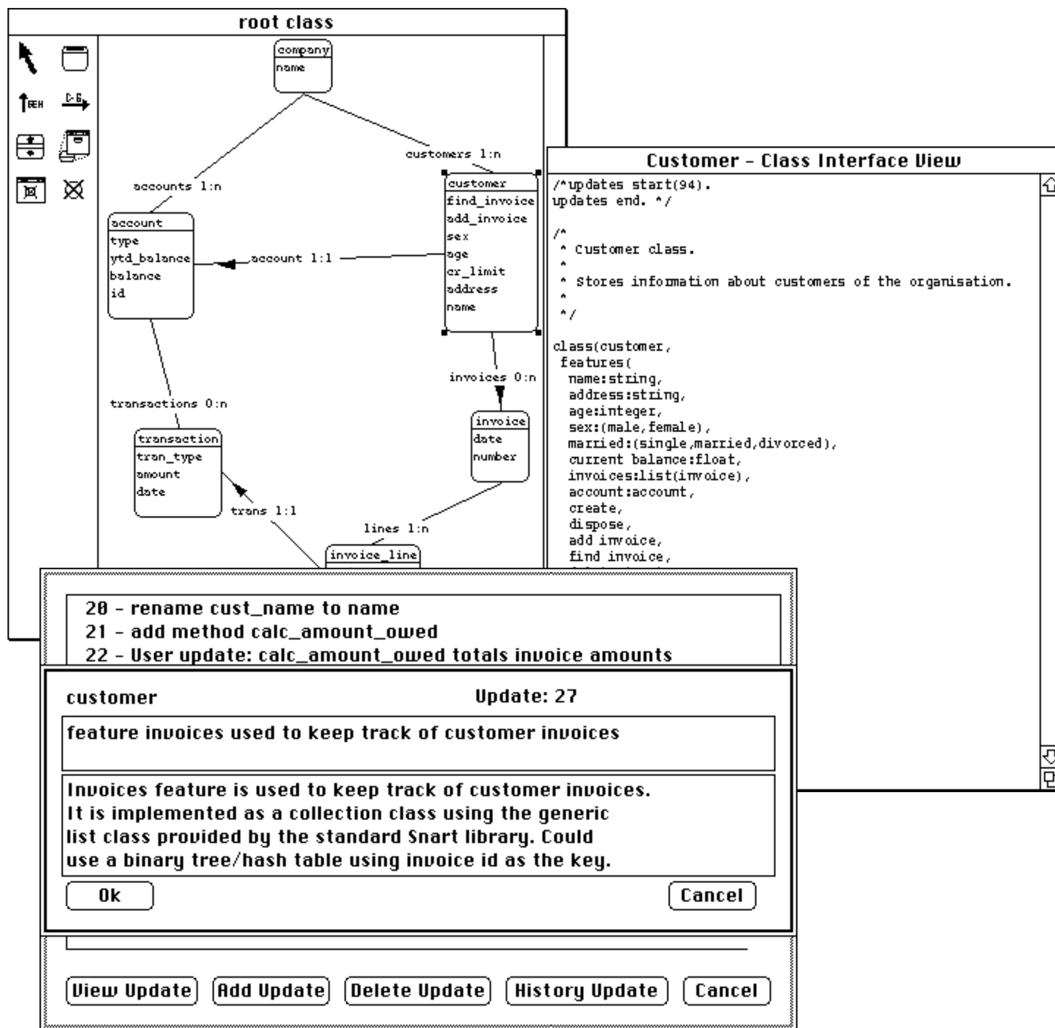

Figure 5. Example of a user-defined change description from SPE.

Semantic inconsistencies can be described both by highlighting affected view information and by presenting detailed change descriptions to users. It is often harder to fully describe the cause of semantic inconsistencies, and in most situations impossible to provide automatic resolution facilities. Often an environment can only inform users of the editing change(s) that caused the inconsistency and the semantic error(s) that now exist. SPE allows users to interact with semantic inconsistencies to find the source of the inconsistencies (usually individual view edits) and to obtain detailed descriptions of their causes.

Users can create their own "change descriptions" via modification history dialogs, as shown in figure 5. These do not directly represent any particular view inconsistency, but rather serve as user-defined view documentation. Such change descriptions are associated with specific information items, and, like structural and semantic change descriptions, are propagated to other views of these items. They also serve a useful purpose in supporting context-dependent communication in cooperative environments, allowing cooperating users to interact via "messages" sent via the change description mechanism [11, 13]. All change descriptions can be annotated via the dialog shown in figure 5, allowing users to specify additional reasons why changes have been made.

3.4. Highlighting Inconsistencies

It is often appropriate for an environment to present inconsistencies in a more context-dependent way than via textual change descriptions. Figure 6 shows a screen dump from MViewsDP, an interface builder which supports graphical and textual dialog specification. A dialog is under design, and the dialog control button 'Ok' in the graphical view has been shifted. This is reflected in the textual view header by a change description. This change description can also be shown in a pop-up menu associated with the graphical view.

There is, however, a semantic inconsistency resulting from the Ok button's border overlapping that of its enclosing dialog box. This must be resolved, as the specification is currently invalid. This inconsistency is indicated by shading the Ok button icon, so the user of the environment is drawn to this component. The change description is shown in a dialog if the user requests to see it, in order to describe the inconsistency.

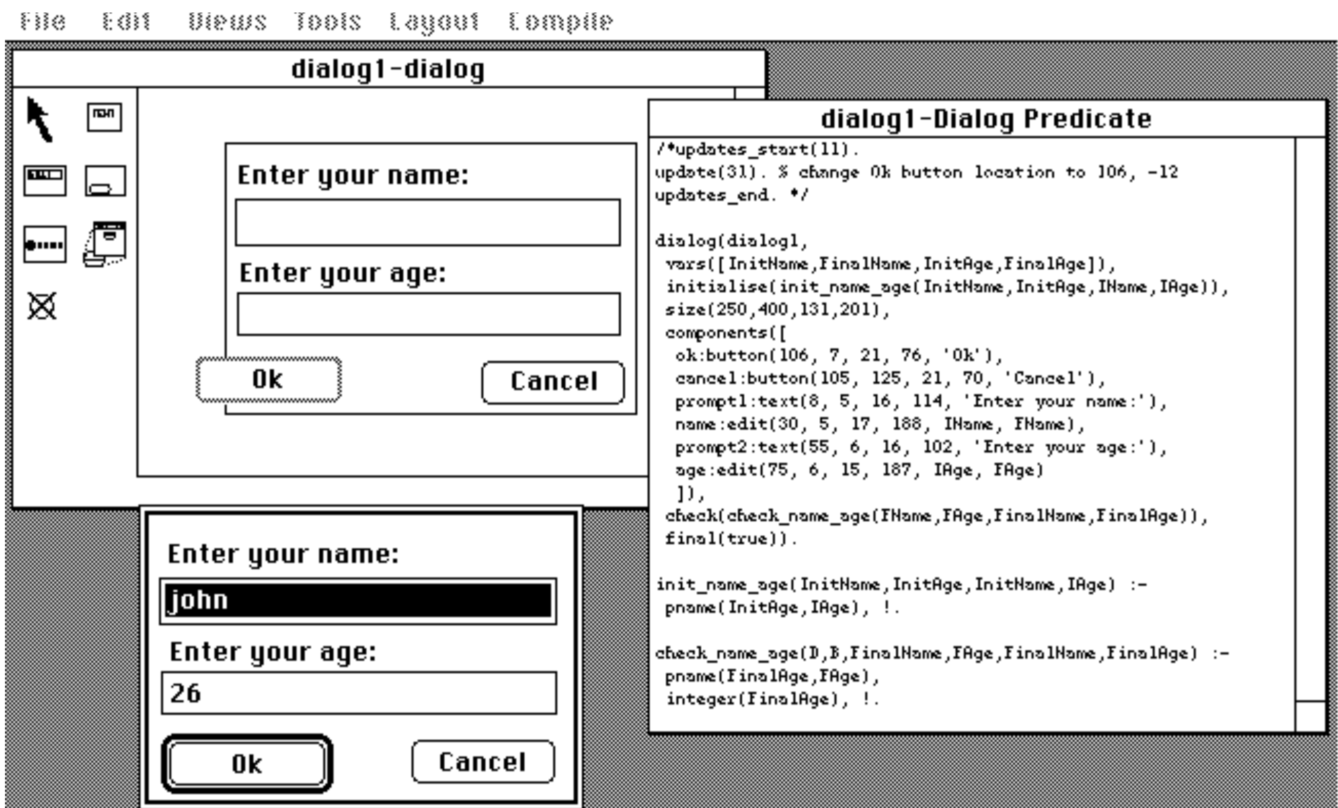

Figure 6. Indicating inconsistencies in graphical views.

A variety of presentation techniques can be utilised to highlight inconsistencies in a particular part of a view. These include shading and colouring graphical icons, changing font, style, size or colour of text, or more dynamic techniques, such as blinking affected icons. Highlighting inconsistencies and explicit description

of inconsistency via change descriptions are complementary techniques. The presence of highlighting in a view can draw the user's attention to a particular aspect, which may then be interacted with to view detailed change descriptions.

3.5. Presenting Stored Inconsistencies

It is often useful to retain descriptions of inconsistencies and present these to users on request in many environments. Figure 7 shows a screen dump from OOEER, an environment supporting integrated OOA/D and EER modelling [14]. OOEER was produced by integrating SPE and MViewsER, an environment supporting integrated graphical EER and textual relational schema modelling. When an EER notation view is modified, the update is translated into an update on corresponding OOA/D views. Some translations can be automatically applied by OOEER, others result in partial translations with view inconsistency. These inconsistencies are presented by colouring affected icons and text. The user can then view a detailed description of the inconsistency in a dialog.

The dialog in figure 7 describes changes made in the EER view customer entity which affect the OOA/D view customer class. Items highlighted with a '*' were actually made in the EER view and translated into OOA/D view updates. The user can make further changes to the OOA/D view if the EER update could only partially be translated into an OOA/D view update. For example, adding a EER relationship (change #8) was translated into the addition of an OOA/D association relationship (change #9). The user then refined this relationship to an aggregation relationship (change #10). This couldn't automatically be done, as the EER notation does not support the distinction between different kinds of relationships.

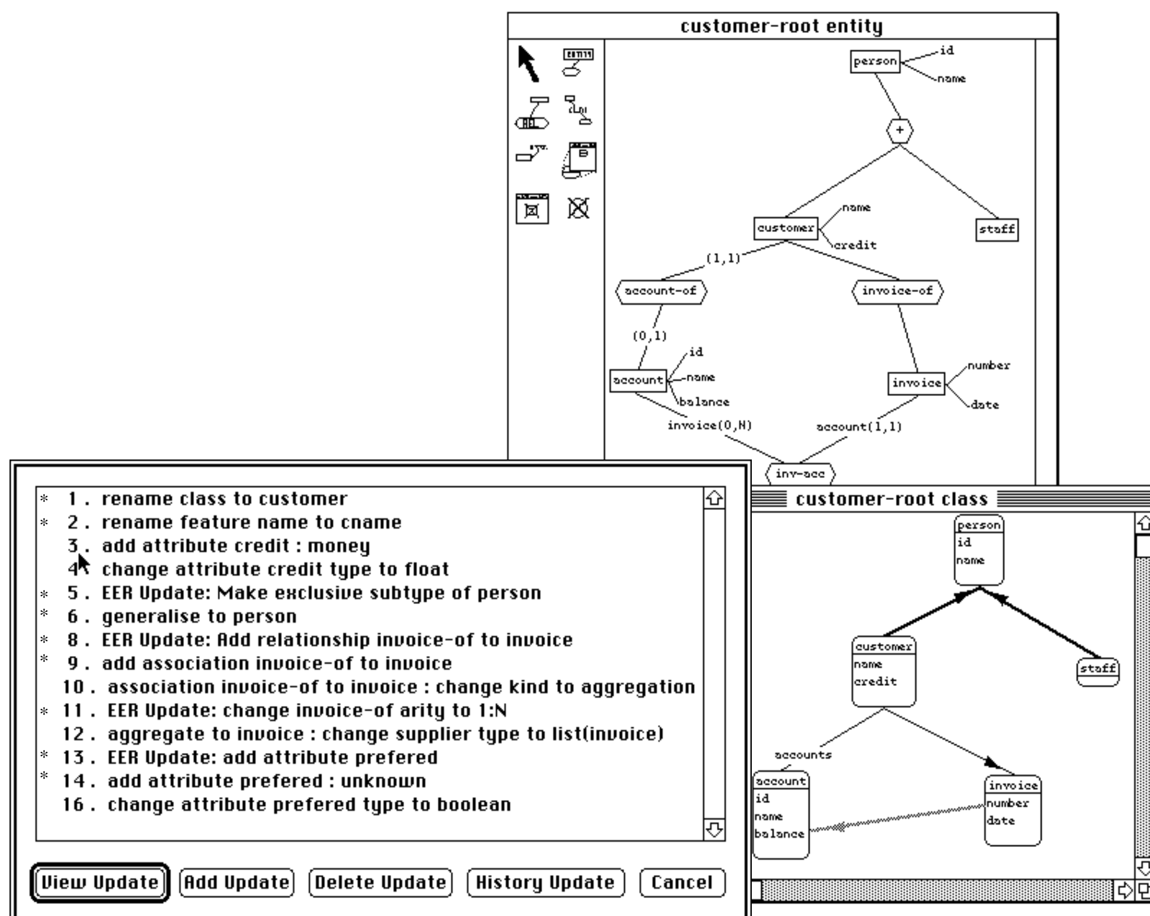

Figure 7. Graphical view inconsistencies.

OOEER retains inter-notation view change descriptions as a "modification history", documenting the history of updates a view or view component has undergone. OOEER provides menu options to locate inconsistent views or components, making it easier for users to determine what inconsistencies need their attention. On display, OOEER highlights which items in a view are (or may be) inconsistent.

4. Interaction with Inconsistencies

The presentation of inconsistencies is not sufficient to support flexible multi-view editing of shared information. Environments must allow users to interact with these inconsistencies to resolve them. Our approach provides a range of options for users and environment implementers to support interaction with inconsistency presentations. These include: allowing users to select presentations and request the environment to automatically resolve the inconsistency; presenting users with a list of optional updates which can resolve the inconsistency; and requiring users to manually update the affected view. Our techniques facilitate cooperative work by broadcasting and presenting change descriptions to other users and allowing them to interact with these to resolve interperson view inconsistencies.

4.1. Selection and Action of Descriptions

Change descriptions can be selected by users and a request made to the environment to resolve the inconsistency described. Users interactively select change descriptions in a dialog or in a view, issue a request to implement the update, and are informed of the result of this request. Figure 8 shows an example of requested view updates in SPE. The user has selected all of the change descriptions in the textual view header and asked SPE to update the view's text. The first two changes can be automatically applied by the environment and results in the attribute address being renamed to address1, and the addition of attribute address2. Both of these change descriptions are deleted to indicate successful update of the view. The other two changes, however, can not be automatically applied, and the change descriptions are left in the view's text to indicate this. Updates selected in a dialog but which could not be successfully applied by the environment are highlighted or shown to the user in another dialog.

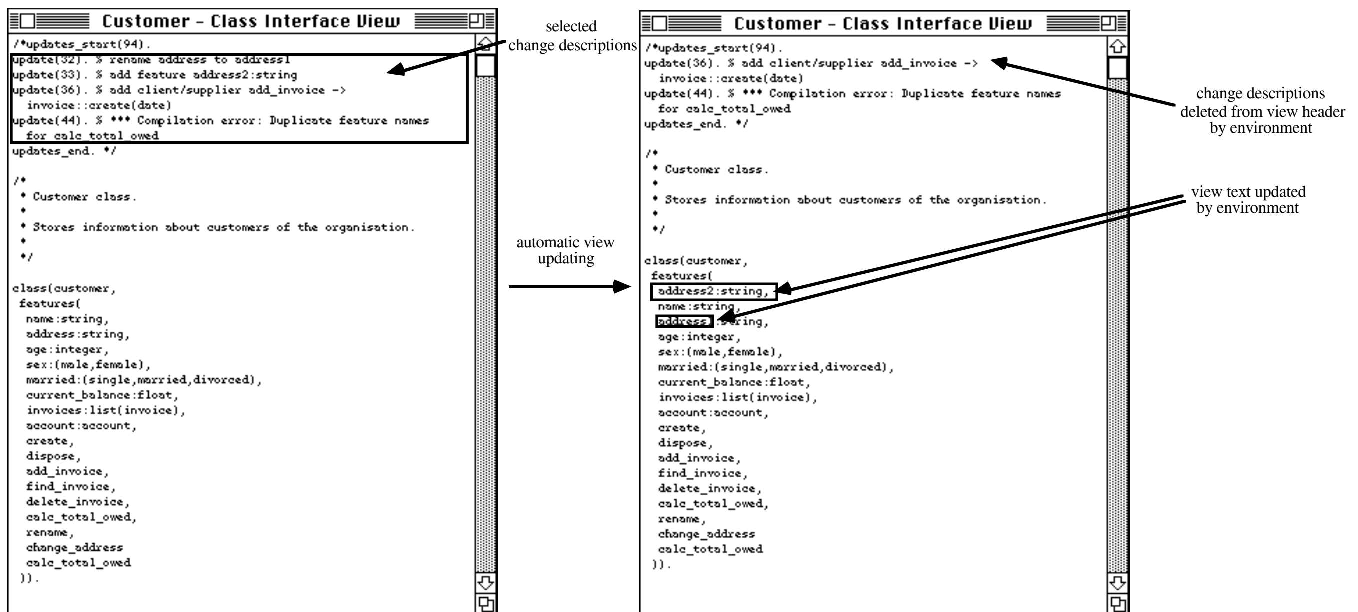

Figure 8. Automatic resolution of inconsistency.

View changes can be animated in sequence, if desired, so the user can "see" the effects of a sequence of changes. Updated items in the view can also be distinguished, for example by shading, colouring and font characteristics, allowing users to identify aspects of the view which were updated. This helps users determine the effects of changes and, moreover, to ensure that the desired view update(s) were inferred correctly by the environment.

It is usually straightforward to update graphical view information by actioning change descriptions. Environment implementers specify change description patterns and action sequences to perform on the appropriate graphical view components. As our textual views are free-edited, and hence the structure of the view information is not represented except in the view's text, this information is automatically kept consistent via a technique we call incremental unparsing. A view is incrementally parsed to identify where text should be added, updated or deleted. Change descriptions are then unparsed, or "pretty printed", into the view by modifying the affected text as appropriate. Further details on the implementation of this technique can be found in [12].

4.2. Selection of Optional Update

Often an inconsistency arises when a multi-view editing environment can not choose between several possible options for resolving an inconsistency. This occurs in environments which translate changes between views of information of differing levels of abstraction or between views which only share part of the information being viewed. Examples are OOEER, which translates changes to and from EER and OOA/D views of the same software system, and SPE, with informal code views and formal Object-Z views. Often in these environments a change made to one notation view can be partially translated into a change in views using the other notation. The user must complete the translation by choosing from a range of possible view updates.

For example, in OOEER if a relationship is added in an EER view between customer and invoice entities, then this is translated by the environment into the addition of a relationship between the customer and invoice classes in the OOA/D views. However, more information about the relationship is needed in the OOA/D views: the relationship might be an association or aggregation OOA relationship (not specified in the EER view), or if it is a OOD client/supplier relationship, it may specify a method call between objects and thus need to specify caller/called method names and arguments. As this information is not specified in the EER views, OOEER defaults the relationship to an OOA association relationship, and lets the user refine this further by changing it to an aggregation or client/supplier relationship and adding extra information about it. OOEER indicates the new relationship has been defaulted by greying it, and provides a pop-up menu to allow the user to easily change its type. The user can also optionally specify more information about the new relationship in the OOA/D views as appropriate. Figure 9 shows an example of such an interaction with an inconsistency via optional update.

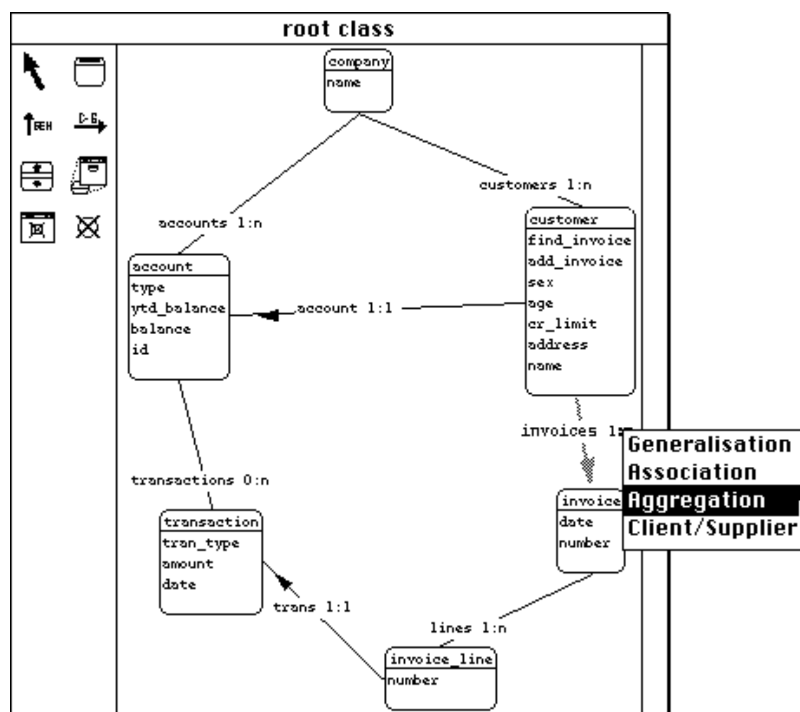

Figure 9. Selection of optional update to fully resolve an inconsistency.

4.3. Manual Resolution of Inconsistency

In this approach, the user must be prompted to manually update a view to resolve an inconsistency. The environment can assist the user in identifying parts of the view which are inconsistent and provide a description of the view updates causing the inconsistencies using the techniques of Section 3. Such a situation usually occurs between views which provide quite different representation, interaction or abstraction levels on shared information.

For example, many design-level updates in SPE can not be sensibly defaulted in textual code-level views. The design-level view updates are presented to users as change descriptions and it is left to the user to manually resolve the view inconsistency. The change descriptions can then be deleted by the user when they are no longer required. The addition of a design-level client/supplier relationship is usually

implemented as a code-level method call. Such a change can not be automatically implemented in a code-level textual view, as the environment does not know the appropriate method arguments (variables or constants) and position of the method call within the method code. Semantic errors require users to correct the problem and can not usually be automatically corrected by the environment. Figure 10 shows an example of this manual inconsistency resolution.

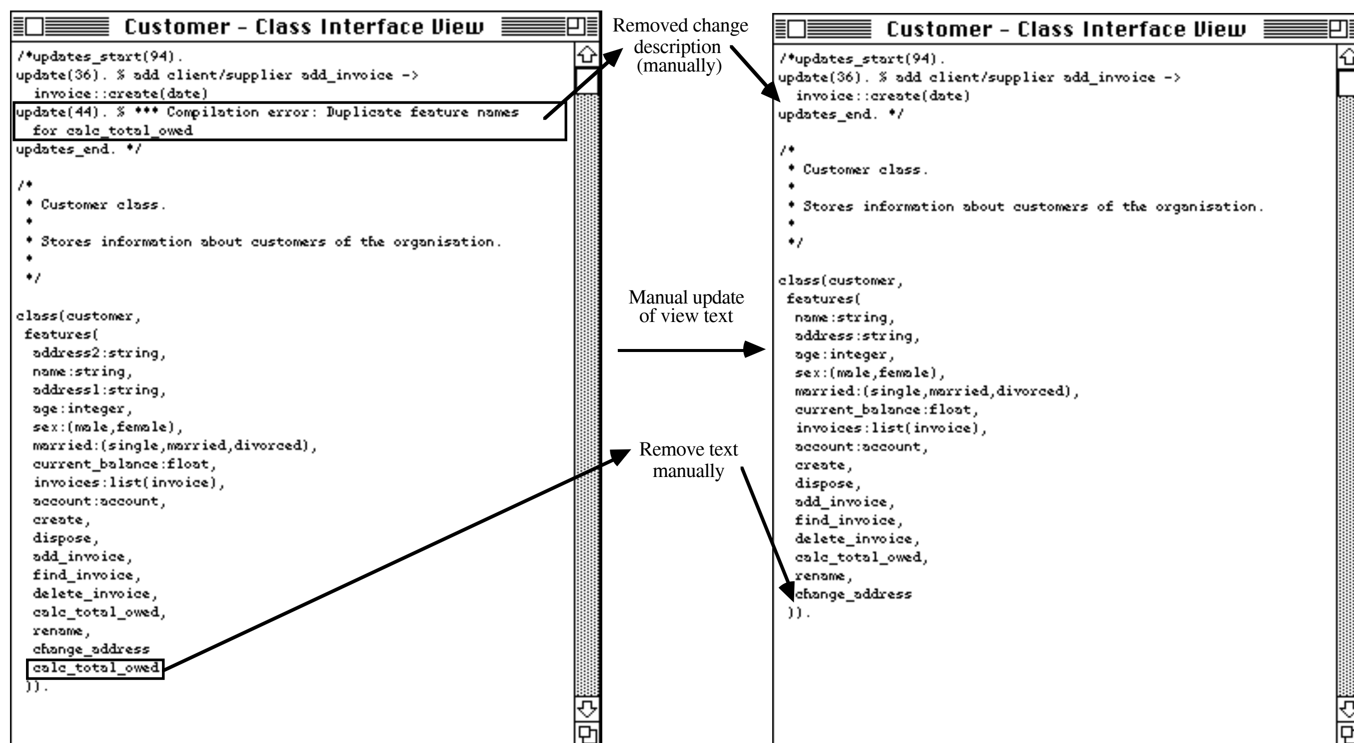

Figure 10. Manual resolution of inconsistency.

In addition to resolving inconsistencies, users can interact with inconsistency representations in other ways. An environment can add interaction "hot-spots" to highlighted, inconsistent view items or to change descriptions. Hypertext buttons can be clicked on to: display change descriptions in dialogs or pop-up menus; display pop-up menus with a list of optional updates; or display of the view whose update caused the inconsistency.

4.4. Versioning and Cooperative Work via Inconsistency Presentation and Interaction

Computer-Supported Cooperative Work (CSCW) systems may utilise a multi-view editing approach to sharing and modifying information [32, 36, 29]. Inconsistencies between views become more difficult to resolve as different users of the views are affected. We have used our technique to provide support for asynchronous, semi-synchronous and synchronous multi-view editing for C-SPE, a collaborative form of SPE [11].

Change descriptions are used to describe changes between different versions of updated views i.e. together form a modification history for views. By using these modification histories as versions, a flexible multi-view version control system can be produced [11]. A non-sequential undo/redo facility can be provided by reversing or reapplying view changes, and this can be extended to supporting version merging and revision [11]. An example from C-SPE is shown in figure 11, with conflicts between two merged versions presented to the merging user as a sequence of merge conflict change descriptions. Semantic conflicts produced by merging are presented in the same manner.

Inconsistencies between multiple views shared by different users are presented in similar ways to those described previously. Asynchronous collaboration is supported by giving users their own versions of each view, with a single developer merging these versions at a later time. Semi-synchronous collaboration is supported by broadcasting change descriptions between different users' environments as they occur. These are presented to users by being stored or presented in a view or in a dialog. Figure 12 shows an example of semi-synchronous view editing in C-SPE. Synchronous collaboration requires users to share and edit the same version of a view, with fine-grained locking on view components [11].

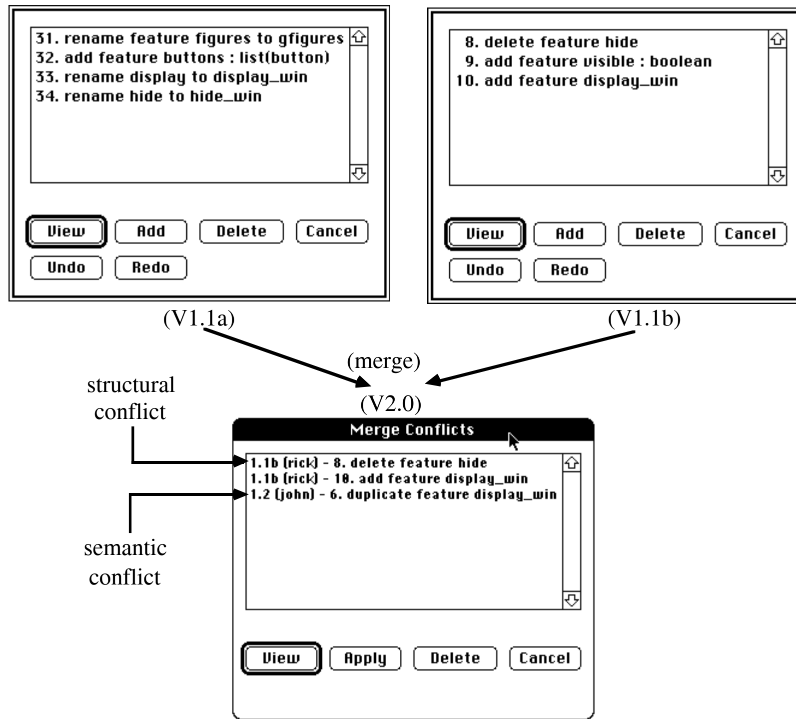

Figure 11. Version merging and presentation of merge conflicts via change descriptions.

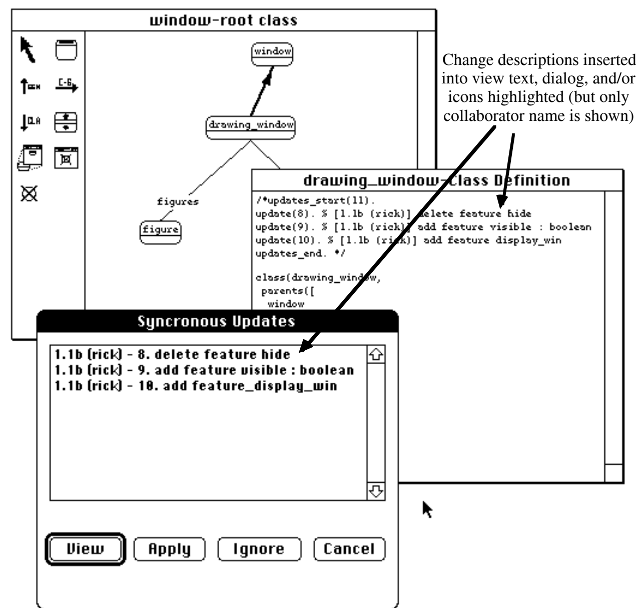

Figure 12. Semisynchronous view editing via change description broadcasting.

We are currently experimenting with multi-view editing facilities which utilise inconsistency presentation and interaction to coordinate cooperative work and planning processes. This allows users to specify the action their environment should take when generating or receiving change descriptions. This includes specifying the method of obtaining and/or presenting extra information about the view update made, such as the work or planning context it was made in, and specifying action to take when processing different kinds of view edits. This allows users to build up highly flexible and configurable cooperative systems on top of multi-view editing environments which support our view consistency model [15].

5. A Supporting Software Architecture

We describe a basic architecture for supporting our inconsistency presentation and interaction techniques. We have used this model as the basis for an object-oriented framework of classes, which are reused to construct new multi-view editing environments.

5.1. CPRGs

Our view consistency model requires descriptions of the changes made to a view to be generated and propagated to other views which share the updated information. We have developed a software architecture, called Change Propagation and Response Graphs (CPRGs), that supports a wide range of flexible consistency management mechanisms based on change description propagation [13]. The kernel idea of CPRGs is that information is structured as attributed *components* linked by inter-component *relationships* , forming a graph-based description of related information items. The state of a component is modified by *operations*, which generate *change descriptions* describing the modification the updated component has undergone.

Change descriptions are propagated to all relationships a component participates in. These relationships respond to change descriptions by: updating their own or related component states; passing on change descriptions to related components; or ignore the change to the updated component. This technique supports a wide variety of consistency management facilities used by ISDE environments, including flexible multiple view consistency, inter-component constraints, efficient incremental attribute recalculation, generic undo/redo facilities, and version control and collaborative facilities.

Figure 13 illustrates how consistency management in SPE is supported by the CPRG model: 1) a class component is renamed, generating a change description of the form update(class,name,customer,customer2); 2) this change description is propagated to all relationships the class component participates in; 3) the relationships determine if other components related to class need to be updated. If they do, they either apply operations to update these components or forward the change description to them; 4) the related components are updated to maintain view consistency, inform users of inconsistencies that can't be automatically resolved, or inform cooperating users of view changes.

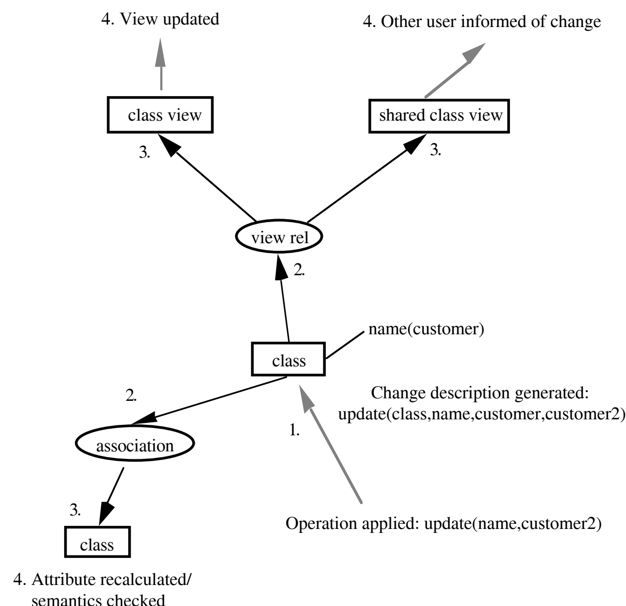

Figure 13. Example of consistency management using CPRGs.

In addition, our CPRG model supports the composition of discrete change descriptions into *macro* change descriptions [13]. It is often more useful to store and present to users the more abstract macro change description rather than all of its individual change descriptions, reducing the number of individual change descriptions presented. CPRG macro change descriptions can be presented to users in the same manner as simple change descriptions, but an environment can present or allow users to interact with a macro description in more abstract ways than its composite descriptions. As the components of macro change descriptions are still retained, the user can request to see them if necessary.

5.2. MViews

In addition to developing the multi-view editing inconsistency generation, propagation, presentation and interaction techniques described above, we have developed MViews, a reusable object-oriented framework for building environments using these mechanisms. MViews classes are specialised to build new multi-

view editing environments [9, 12]. MViews is based on the CPRG model, and provides a variety of classes for implementing base information representations, multiple textual and graphical views, and multi-view editors. Figure 14 illustrates the structure of SPE, built using MViews. A *base layer* acts as the shared information repository for views. A *view layer* defines the contents of each view in terms of CPRG components and relationships i.e. views have the same structure as the base information items they represent. A *display layer* provides a graphical or textual rendering of the view layer, and provides suitble editing facilities for this representation. External tools not built using MViews can also be interfaced at the view and display layers.

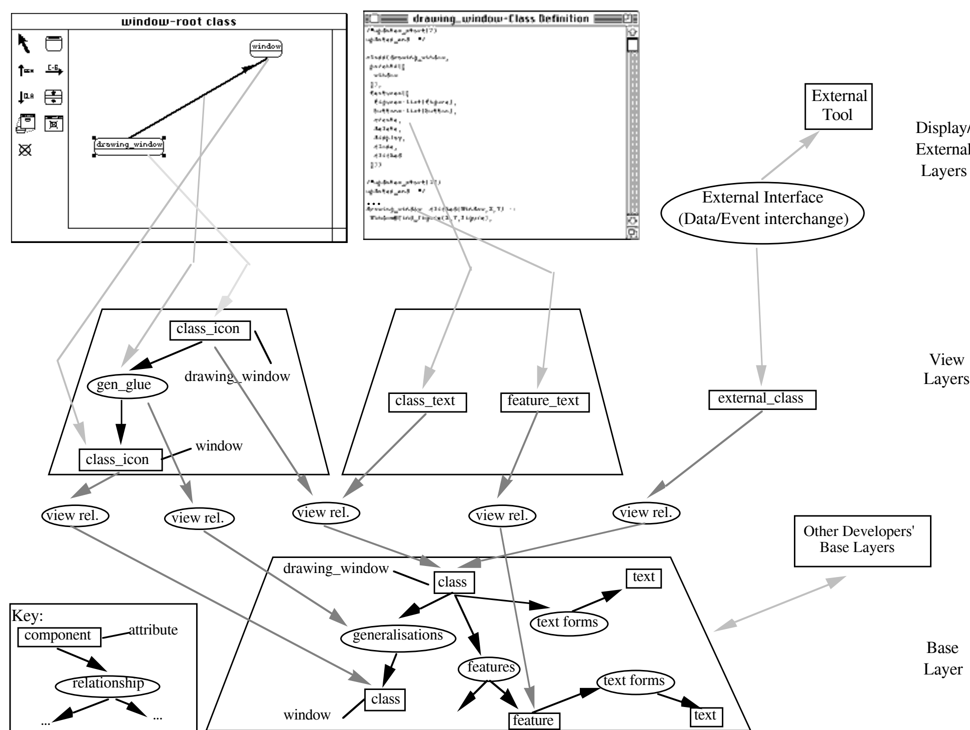

Figure 14. An architecture supporting change description propagation.

MViews uses the inconsistency presentation and interaction techniques described previously to support flexible multi-view editing, as shown by the SPE examples described earlier. Change descriptions are retained so that when users interact with inconsistency presentations, MViews can determine the nature of the inconsistency. When a change description representation is selected and/or interacted with in a textual view or dialog, MViews can determine the CPRG change description being interacted with. The environment can then take action for the view inconsistency being interacted with. CPRG change descriptions, such as update(Class,class_name,OldName,NewName), are translated into readable forms, such as 'rename class Class to NewName', by environment implementers specifying change description patterns and resultant formatted representations. Change descriptions generated in other views which are not relevant to the user of another view sharing the updated information are filtered out during propagation to the affected view.

5.3. Implementation

MViews is implemented in Snart, an object-oriented extension to Prolog [10], running on Macintoshes using LPA MacProlog[1]. Environment implementers specialise Snart classes to define new environment information repositories, multiple views, and view renderings and editors. Snart is a persistent language, with objects dynamically saved and loaded to a persistent object store, making information repository and view persistency management transparent for environment implementers.

Change descriptions are represented as Prolog terms. This makes change description generation, propagation and storage very efficient, resulting in an architecture which performs well, even for large

[1] A C++ implementation of MViews is currently being developed

applications such as SPE. Representations of these terms are generated for display to users via declarative Prolog predicates. Being Prolog terms, change descriptions can be stored in Prolog lists to store inconsistency representations and form modification and version histories. They are converted to and from a textual representation for broadcasting to other users' environments to support cooperative work [16].

MViews textual view editors are standard Macintosh free-edited text windows. Definite Clause Grammar (DCG) parsers are used to convert textual view contents into Prolog terms, used to determine base view component updates. Graphical view components are rendered and edited using LPA MacProlog's Graphics Description Language (GDL) facilities. This allows icon appearances to be easily modified to incorporate shading and/or colouring, as well as scaling and various other translations. The font style of any text associated with items can also be modified for inconsistency presentation.

6. Experience

We have used the techniques described in this paper in several multi-view editing environments. Some of these environments are briefly described below. A brief discussion of users' perceptions of the techniques, some disadvantages we have encountered, and a comparison with existing environments and techniques are given.

6.1. Environments

ISDEs built using the techniques described in this paper include SPE [11], EPE [1] and C-SPE [16]. EPE provides graphical EXPRESS-G views and textual EXPRESS views, and is used for product modelling design applications. C-SPE extends SPE to support collaborative software development using synchronous, semi-synchronous and asynchronous interaction techniques. SPE has recently been extended to incorporate textual Object-Z formal specification views [17], which are kept consistent with SPE implementation textual views and design graphical views by expanding change description representations.

Modelling environments and interface builders include MViewsER [14], MViewsDP [13], OOEER [14], and NIAMER [40]. MViewsER provides graphical views of ER data models and textual views of relational schema. MViewsDP provides graphical views of dialog specifications and textual views of dialog constraints. OOEER integrates SPE and MViewsER to provide an environment which supports truely integrated OOA/D and EER views. Changes to OOA/D views are propagated to EER views, and vice-versa, helping designers keep views of a common model consistent. NIAMER integrates MViewsER and MViewsNIAM, an environment which supports modelling using the NIAM notation. Changes to NIAM views are propagated to EER views, and vice-versa, helping designers keep views of a common model consistent.

Visual programming environments include ViTABaL [18], Skin [20], and HyperPascal [26]. These environments provide graphical views for specifying systems based on the tool abstraction paradigm (ViTABaL), flexible user interface components (Skin), and a visual Pascal (HyperPascal). Graphical views are kept consistent by representing inconsistencies via icon highlighting and change description representation in dialogs. Low-level textual views are kept consistent with aspects of the graphical views via change description expansion.

6.2. User Perception

Users of these environments like having view inconsistency representations to see and interact with [12, 11, 13]. When moving to a view, users are always informed of updates which potentially affect the view. Use of an appropriate way of presenting inconsistencies gives users immediate feedback on the accuracy and consistency of their views. For complex views, the presence of inconsistency presentations assists users in determing where in the view inconsistencies exist and how to go about resolving them. Hypertext facilities to navigate to related views (also supported by MViews) allow users to quickly navigate between inter-dependent, inconsistent views, allowing them to quickly resolve inconsistencies. Our techniques work equally well for small environments with relatively simple views, such as MViewsER and MViewsDP, and scale up for larger environments with many complex views, such as SPE, OOEER, NIAMER and EPE.

Environment implementers choose between using unobtrusive techniques to inform users of view inconsistencies, such as auto-expansion into textual views and highlighting, to presenting change description representations in dialogs immediately a view is selected. As automatic update of graphical and textual views is often possible, implementers can assist users by having an environment resolve

inconsistencies whenever it can. We have found that it is often useful to still present representations of change descriptions, or highlight affected view components, even when inconsistencies have been sucessfully resolved. This allows users to confirm a correct modification has been done and these act as modification history documentation.

The main disadvantages of our approach have been encountered when environment implementers choose an inappropriate way of presenting or interacting with inconsistencies. For example, if a view is not used for some time, or a view is often affected by many other view updates, automatic expansion of change description representations into the view can result in large numbers of change descriptions, and users can become confused. It is more appropriate in these situations to highlight affected view components and indicate that change descriptions can be browsed in a dialog. This problem is compounded if an environment does not allow users to choose their own preferred mechanism for presenting or interacting with inconsistencies. Ideally environments should allow users to configure the view consistency management process.

It can be frustrating for users if some view inconsistencies can't be automatically resolved, when the user knows there is a simple resolution process the environment should carry out. This is a problem when the environment implementer does not know about this process, or the process is specific to a particular user. Once again users should be able to extend the environment's automatic resolution algorithms.

6.3. Comparison to Existing Approaches

Our inconsistency presentation and interaction techniques allow multiple, editable graphical and textual views of information to be automatically or partially kept consistent, with support for inconsistency management. Most other multi-view editing environments and models, such as the MVC [24], the ItemList [6], and Zeus [4], are not as flexible as they do not have inconsistency presentation. Our approach is useful where information shown at differing levels of abstraction must be kept consistent, as it supports "fuzzy" consistency management, where changes to one view impact on others so the environment can not automatically resolve inconsistencies. Other systems supporting multiple textual and graphical views of the same information, such as Dora [32] and PECAN [33], can not propagate such changes between views. Controlled inconsistency management between views is supported by retaining change descriptions denoting the inconsistencies, and users of the environment then determine when the inconsistencies are to be resolved. This a simple, elegant approach to inconsistency management, less cumbersome than the logical expressions denoting inconsistencies of [8].

Version control and cooperative work facilities are supported, as change description lists form a natural modification history for views and their components, and can be divided into versions. Broadcasting change descriptions and utilising appropriate presentation and interaction techniques facilitates synchronous and semi-synchronous cooperation that other cooperative environments, such as GroupKit [38], Mjølner [28], [31], and Mercury [22], do not support.

Our techniques are not confined to MViews-style free-edited view environments.The same problem of high-level view updates propagated to low-level views, and vice-versa, occurs in most multi-view editing environments. Structure-oriented environments, such as Dora [32], Mjølner [27], and the Cornel Program Synthesizer [37], could utilise these techniques when trying to maintain consistency between their views of information at differing levels of abstraction. In fact, automatic resolution of inconsistencies becomes easier in these environments, as the underlying structure of both graphical and textual views is readily accessible [2].

7. Summary

To achieve true consistency management in multi-view editing environments <u>all</u> view modifications must be propagated to other views which are affected by the change. Unlike existing environments and models, our approach supports full view consistency management by presenting users with representations of all view inconsistencies. Presentation mechanisms include descriptions of changes in dialogs, pop-up menus and textual views, view component highlighting, and lists of inconsistency resolution options. Users can interact with these inconsistencies by requesting automatic inconsistency resolution, choosing a resolution option, and manual resolution. Our technique also supports non-structural semantic and user-defined view inconsistencies, view versioning and cooperative work facilities, and the grouping of hierarchical, macro inconsistency descriptions. These techniques can all be efficiently implemented and packaged for reuse.

Use of environments which exhibit these view consistency techniques indicates users prefer these environments to those which do not support inconsistency presentation and interaction.

We are extending many of our environments to support user-definable links between view components. These allow information items to be linked in arbitrary, user-defined ways, with the level of consistency management across these relationships mostly restricted to change description display and inconsistency highlighting. These will allow users to link information items in ways not conceived of by the environment designers, and yet allow the multi-view editing environment to maintain at least a basic level of (in)consistency management between them. In order to make multi-view editing environments more suitable for different users, environments need to support user-configurability. This includes the capability to allow users to specify their own preferred inconsistency presentation, interaction and automatic resolution techniques, both for single-user environments and multi-user, cooperative work environments.

References

1. Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C. Directions in modelling environments. *Automation in Construction* , 4 (1995), 173-187.

2. Arefi, F., Hughes, C.E., and Workman, D.A. Automatically Generating Visual Syntax-Directed Editors. *Communications of the ACM 33*, 3 (March 1990), 349-360.

3. Avrahami, G., Brooks, K.P., and Brown, M.H. A Two-View Approach to Constructing User Interfaces. *ACM Computer Graphics 23*, 3 (1990), 137-146.

4. Brown, M.H. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages,* IEEE Computer Society Press, 1991, pp. 4-9.

5. Cox, P.T., Giles, F.R., and Pietrzykowski, T. Prograph: a step towards liberating programming from textual conditioning, , IEEE Computer Society Press. In *Proceedings of the 1989 IEEE Workshop on Visual Languages,* 1989, pp. 150-156.

6. Dannenberg, R.B. A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors. *Software-Practice and Experience 20*, 2 (February 1990), 109-132.

7. Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some Issues and Experiences. *Communications of the ACM 34*, 1 (January 1991), 38-58.

8. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Ssoftware Engineering 2*, 8 (August 1994), 569-578.

9. Grundy, J.C. and Hosking, J.G. A framework for building visusal programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1993, pp. 220-224.

10. Grundy, J.C. *Multiple textual and graphical views for Interactive Software Development Environments*, Ph.D. dissertation, University of Auckland, Department of Computer Science, June 1993.

11. Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. Connecting the pieces, *Visual Object-Oriented Programming,* Prentice-Hall (1994), Chapter 11.

12. Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with MViews," Working Paper, Department of Computer Science, University of Waikato, 1994.

13. Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Supporting flexible consistency management via discrete change description propagation. to appear in *Software - Practice and Experience*.

14. Grundy, J.C. and Venable, J.R. Providing Integrated Support for Multiple Development Notations. In *Proceedings of CAiSE'95,* LNCS 932, Springer-Verlag, Finland, June 1995, pp. 255-268.

15. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. Coordinating, capturing and presenting work contexts in CSCW systems. In *Proceedings of OZCHI'95,* Wollongong, Australia, Nov 28-30 1995, pp. 146-151.

16. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Amor, R. Support for Collaborative, Integrated Software Development. In *Proceeding of the 7th Conference on Software Engineering Environments,* IEEE CS Press, April 5-7 1995, pp. 84-94.

17. Grundy, J.C., and Hosking, J.G. Support for Integrated Formal Software Development. In *Proceedings of APSEC'95,* Brisbane, Australia, December 1995, IEEE CS Press, pp. 264-273.

18. Grundy, J.C. and Hosking, J.G. ViTABaL: A Visual Language Supporting Design By Tool Abstraction. In *Proceedings of the 1995 IEEE Symposium on Visual Languages,* IEEE CS Press, 1995, pp. 53-60.

19. Hart, R.O. and Lupton, G. DECFUSE: Building a graphical software development environment from Unix tools. *Digital Tech Journal 7*, 2 (1995), 5-19.

20. Hosking, J.G., Fenwick, S., Mugridge, W.B., and Grundy, J.C. Cover yourself with Skin. In *Proceedings of OZCHI'95,* Wollongong, Australia, Nov 28-30, 1995, pp.

21. Kaiser, G.E. and Garlan, D. Melding Software Systems from Reusable Blocks. *IEEE Software 4*, 4 (July 1987), 17-24.

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

23. Kiper, J.D. A framework for characterisation of the degree of integration of software tools. *Journal of Systems Integration 4* (1994), 5-32.

24. Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (1988), 8-22.

25. Linton, M., Vlissides, J.M., and Calder, P.R. Composing User Interfaces with InterViews. *COMPUTER 22*, 2 (1989), 8-22.

26. Lyons, P., Simmons, C., and Apperley, M. HyperPascal: Using visual programming to model the idea space. In *Proceedings of the13th New Zealand Computer Society Conference,* Auckland, August 1993, pp. 492-508.

27. Magnusson, B., Bengtsson, M., Dahlin, L. An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development. In *Proceedings of TOOLS '90,* Prentice-Hall, 1990, pp. 635-646.

28. Magnusson, B., Asklund, U., and Minör, S. Fine-grained Revision Control for Collaborative Software Development . In *Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering,* Los Angeles CA, December 1993, pp. 7-10.

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

30. Narayanaswamy, K. and Goldman, N. Lazy consistency: a basis for cooperative software development. In *1992 ACM Conference on Computer-Supported Cooperative Work,* ACM Press, 1992, pp. 257-264.

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

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

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

34. Reiss, S.P. GARDEN Tools: Support for Graphical Programming. In *Proceedings of Advanced Programming Environments,* LNCS 244, Springer-Verlag, Trondheim, Norway, June 1986, pp. 59-72.

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

36. Reiss, S.P. Interacting with the Field environment. *Software practice and Experience 20*, S1 (June 1990), S1/89-S1/115.

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

38. Roseman, M. and Greenberg, S. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of CSCW'92,* ACM Press, 1992, pp. 43-50.

39. *TurboCASE Reference Manual*, StructSoft Inc, 5416 156th Ave. S.E. Bellevue, WA, 1992.

40. Venable, J.R. and Grundy, J.C. Integrating and Supporting Entity Relationship and Object Role Models. In *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece,* LNCS 1021, Springer-Verlag, 1995.

41. Vlissides, J.M. and Linton, M. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* ACM Press, 1989, pp. 158-167.

42. Wasserman, A.I. and Pircher, P.A. A Graphical, Extensible, Integrated Environment for Software Development. *SIGPLAN Notices 22*, 1 (January 1987), 131-142.

43. Welsh, J., Broom, B., and Kiong, D. A Design Rationale for a Language-based Editor. *Software - Practice and Experience 21*, 9 (1991), 923-948.