Inconsistency Management for Multiple-View Software Development Environments

John Grundy†, John Hosking†† and Rick Mugridge††

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

††Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john, rick}@cs.auckland.ac.nz

Abstract

Developers need tool support to help manage the wide range of inconsistencies that occur during software development. Such tools need to provide developers with ways to define, detect, record, present, interact with, monitor and resolve complex inconsistencies between different views of software artefacts, different developers and different phases of software development. This paper describes our experience with building complex multiple-view software development tools that support diverse inconsistency management facilities. We describe software architectures we have developed, user interface techniques used in our multiple-view development tools, and discuss the effectiveness of our approaches compared to other architectural and HCI techniques.

Keywords: inconsistency management, multiple views, integrated software development environments, collaborative software development

1. Introduction

Software developers work with a variety of specifications of software systems at differing levels of abstraction, including software requirements, analysis, design, implementation and documentation. Inconsistencies between parts of a specification, or between specifications at differing levels of abstraction, can arise during or between phases of software development. Over time, such inconsistencies must be resolved in order to produce a working software system, or partially resolved to produce part of a system for testing and quality assurance purposes. The use of disparate software development tools on a project by multiple developers is usually essential when developing today's complex software systems, but the use of such tools can exacerbate the creation of inconsistencies [44, 53].

Systems have their specifications split among several different tools, often used by different developers, with partial redundancy resulting. Modifying part of a system specification in one tool can thus introduce inconsistencies with related parts of the system specified in other tools, between specifications shared by different developers, or even cause inconsistencies to occur within the same tool. For example, one developer modifying a class interface design in an OOA/D tool will often cause other design diagrams in the same tool to become inconsistent, and cause parts of the system's design held in other tools to become inconsistent. It also may cause code based on these designs to become inconsistent, and the designs to become inconsistent with documentation and analysis diagrams developed by the same or other developers. In a complex system involving multiple developers and development tools, developers are very often unaware of the introduction, or even existence, of such inconsistencies.

The management of inconsistencies during a software development project can be improved when the tools used are tightly, or even loosely, integrated [53, 63, 27, 59]. In such systems, developers interact with multiple "views" (or perspectives) of software at the same and different levels of abstraction. Developers will usually have views partitioned into system requirements, analysis, design, implementation, documentation etc, and further views partitioning specifications at each level of abstraction [29, 63, 53]. Inconsistency detection is needed after one view is edited in order to detect: structural inconsistencies between views (e.g. class attributes added in one view aren't in another); semantic inconsistencies in specifications (e.g. a type mis-match between method calls or a non-existent method is called); inconsistencies between specifications at different levels of abstraction (e.g. system requirements and design conflict); and inconsistencies between the work of multiple developers (e.g. when any inconsistency is caused by different developers working concurrently).

Some inconsistencies may be automatically corrected, for example by tools updating the information in one view when another, related view has been edited. However, many inconsistencies can not, or should not, be automatically corrected. Hence mechanisms are required for tools to inform developers of inconsistencies, and developers require facilities to monitor and resolve inconsistencies. As it is usually impossible to keep a software system consistent at all times, multiple view tools should not be overly prescriptive in attempting to enforce consistency, except when instructed to do so by developers. Tools thus need to support the long-term management of inconsistencies. Having multiple developers adds to the complications of detecting, presenting and tracking inconsistencies made by others, and in negotiating resolutions to inconsistencies.

In this paper we focus on managing inconsistencies primarily between the analysis, design and implementation specifications of software in multiple-view and multiple-user integrated tools. Our main interest and contributions are in detecting structural, semantic, inter-person and inter-process inconsistencies as they occur, and allowing developers to manage these. This contrasts with the kind of "batch" inconsistency checking done by compilers. The contributions of our work include:
- an architecture for software development tools which supports the representation of software specifications, and the definition, detection, representation and propagation of inconsistencies between these specifications
- the realisation of this architecture in frameworks and tool generators for constructing multiple-view and multiple-person software development tools
- techniques for presenting inconsistencies to developers
- techniques that allow developers to monitor, negotiate and resolve inconsistencies
- tool configuration support allowing software developers to configure their environment's inconsistency management policies
- a range of exemplar software development tools built using our architectures that have been deployed on a variety of small- and medium-sized projects and which demonstrate the utility of our techniques.

We begin with various inconsistency problems which can arise in multiple view, multiple user environments. These problems are illustrated with an integrated software development environment that we have developed. We then discuss a range of existing techniques and systems that attempt to address some of these inconsistency management issues in software development tools. Section 4 describes our inconsistency management model, the software architecture that realises this model, and meta-tools we have developed to aid in the construction of software development tools. Section 5 provides a user's perspective on our approaches to the presentation of inconsistency, while Section 6 describes how a user can interact with such presentations to monitor and/or resolve them. Section 7 discusses inconsistency management during collaborative software development. Section 8 argues that inconsistency management configuration facilities are necessary and shows how they can be supported. Section 9 describes our experiences with our inconsistency management techniques and tools, and evaluates them. We conclude with the contributions of this research and directions for future work.

2. Problem Domain: Inconsistencies in Multiple-View Environments

2.1. An Example Multiple-View Software Development Environment

To illustrate the range of inconsistency management requirements of a multiple view software development environment, we introduce SPE (Snart Programming Environment). SPE is an integrated software development environment for developing object-oriented programs [27]. It supports multiple textual and graphical views of information, with full bi-directional consistency management between all views. For example, the same artefact can be viewed in several analysis, design, code and documentation views, as illustrated by the "customer class" specified in Figure 1 in views "root class" (OO design diagram) and view "customer - class Interface" (textual class interface code). We have integrated SPE and the Serendipity software process modelling and enactment environment [34] to support coordinated multiple-user software development in SPE. Figure 1 shows a simple software process for modifying a system design enacted in Serendipity ("aff2. Design, Code & Test-subprocess").
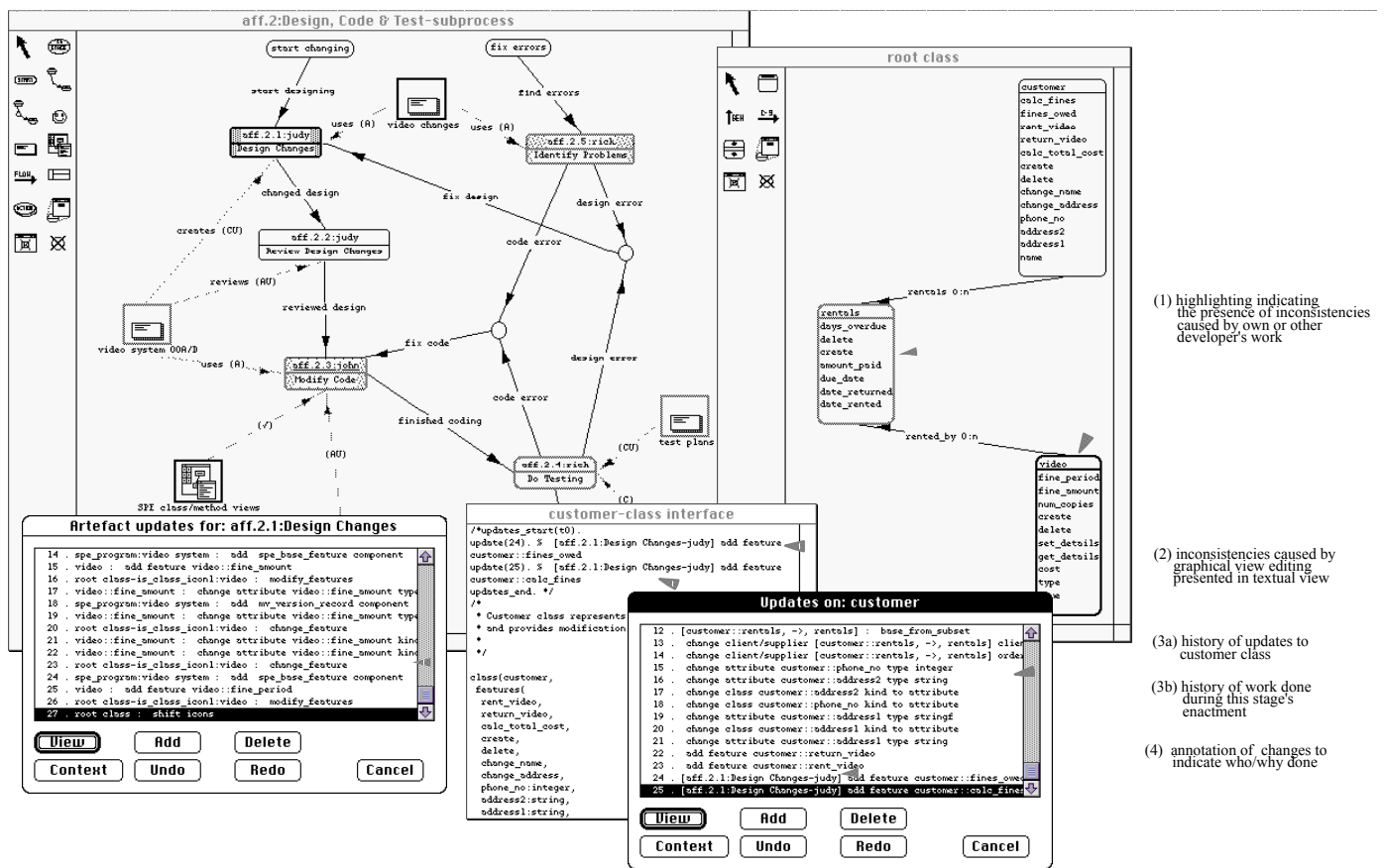

Figure 1.   An example multi-view editing environment.

As information is shown in multiple views in SPE, and at differing levels of abstraction, a range of inconsistencies can occur when modifying different views of software specifications. For example, renaming a class in an OO design view means all other analysis, design, implementation and documentation views become inconsistent unless the class name is changed in these views. Such a change could be carried out automatically by SPE. However, adding a method call connection in an OO design view can not be automatically added to the appropriate textual code view(s) affected by this change, so the code views become inconsistent. SPE has no way of determining appropriate arguments to pass to the method, nor where in the textual code the method call should go. Similarly, changing the type of an attribute in a design view may allow some degree of automatic resolution, but often requires developers to change other design and/or implementation decisions.

When multiple developers share SPE views, they sometimes want to collaborate closely to analyse and negotiate about inconsistencies and how these should be resolved. Thus collaborative editing techniques

are required to facilitate close collaboration and inconsistency management. However, at other times developers will work independently, modifying alternative versions of views. Any inconsistencies generated require tracking, and then negotiation and resolution, usually during version merging.

Figure 1 shows a few examples of how inconsistencies are managed in SPE:

(1)   Icons in views can be highlighted to indicate the presence of inconsistencies.

(2)   Inconsistency "descriptions" can be presented to developers in dialogues or within views to inform them of inconsistencies. Such descriptions may simply describe changes made to other views which affect the specification shown in the view, or may describe semantic constraint violations that need to be resolved. Presentations may also allow developers to "interact" with the inconsistency, for example to select an inconsistency and request more information about it or request a view be automatically modified to resolve the inconsistency.

(3a) "Histories" of changes made to software specifications and

(3b) during process stage enactment are kept, which also record and present semantic inconsistencies (e.g. type mis-matches, undefined variables and methods), and allow tracking of and interaction with inconsistencies.

(4)   Inconsistency descriptions presented to developers can be annotated with extra information e.g. relative importance, additional reasons why it has been detected, and Serendipity process model information.

2.2. Inconsistency Management Requirements

From our experience with developing SPE and many other multiple-tool and multiple-user environments [71,29] we have developed several key requirements for supporting inconsistency management in such tools:

- *Description of syntax and semantics.* The software architecture used to build such an environment must support the representation of a wide range of software specification and view structures and the semantics associated with these structures.
- *Inconsistency detection.* Inconsistencies must be detected when: i) software specifications are modified and related specifications can not be automatically updated to be kept consistent; ii) semantic constraints associated with modified software structures are violated in some way; or iii) the work of multiple developers interferes, causing structural or semantic inconsistencies.
- *Inconsistency representation.* As inconsistencies may range from short-lived to very long-term, ways are needed to represent such inconsistencies, propagate them among related software specifications, and record them for monitoring and later resolution.
- *Inconsistency reason information.* Inconsistencies occur for a reason. Associating the reason for a change can help a user in dealing with that inconsistency, especially when it was made by someone else and/or at a different time. This might simply be that a change has been made to an overlapping software specification [22], and hence the change needs to incorporated in some way in all other affected specifications, or a semantic constraint has been violated, or another developer has made a change. The representation of inconsistencies used by an environment needs to support a description of: what was changed or what constraint was violated; who caused inconsistencies; the process stage or "context" inconsistencies occur in; reasons inconsistencies are detected; and the relative "importance" of inconsistencies, including during what phases of development an inconsistency can be "tolerated" before being resolved.
- *Inconsistency presentation.* Developers need to be informed of the presence of inconsistencies when using views and need to be provided with a variety of information about inconsistencies.
- *Inconsistency monitoring.* Developers need to monitor inconsistencies at different times and in different ways, and thus require facilities to query for specific kinds of inconsistencies and to have these presented appropriately.
- *Inconsistency interaction and resolution.* Developers need to interact with inconsistency presentations to locate their causes, gain more information about them and resolve them.

- *Inconsistency resolution negotiation.* Multiple developers require support for negotiating the resolution of inconsistencies affecting more than one person.
- *Inconsistency management configuration.* Developers require facilities to configure when and how inconsistencies are detected, monitored, stored, presented, and possibly automatically resolved.

3. Related Research

Many software architectures have been developed to aid in building tools that provide multiple views of software development. However, none that we are aware of completely support the range of inconsistency mangement requirements outlined in the previous section.

Database views and active constraint tiggers can be used to build multiple-view systems where the views are informed of changes to model objects and requery the model to update the view's state [53, 1]. For example, MELD [42] and MultiView [1] use a form of database views to support multiple views for software tools. Uni-directional constraint systems, such as Garnet [54], Clock [25], Zeus [12], and those built on top of database management systems, use constraint rules between software specification components which, when triggered, automatically update affected structures or flag the presence of inconsistencies. Multi-directional constraint systems, such as Rendezvous's Abstraction-Link-View [38] and Amulet [55], use more flexible inter-object constraints allowing changes made to repository or view specification objects to maintain view consistency.

All of these trigger and constraint-based approaches use logical constraints and queries over model and view objects, and are thus able to readily detect structural and semantic inconsistencies. They do not, however, represent resulting inconsistencies without using objects or relations to model them, nor are they able to readily associate extra information with inconsistencies. For example, FormsVBT [6], built with Zeus, supports multiple textual and graphical views of user interface specifications and simulations, which are kept consistent under change. While FormsVBT successfully manages to keep views consistent, the authors admit their techniques used do not scale up to more general software specification views, and lack more general inconsistency management facilities [6].

Smalltalk Model-View-Controller [46] and Java-style Observer [24] models support the notion of views of data structure objects, with the ability to propagate objects describing model object changes to observing objects. These "change" objects can be used to represent a range of inconsistencies, and extra information about inconsistencies. However, the architectures lack any application-independent approaches to generating such inconsistency representation objects, and many systems using these architectures simply rely on view objects to reconcile their state to their model objects i.e. using model change notifications as simple event triggers. This severely limits the kinds of inconsistency management that tools can provide to software developers. The ItemList structure [14] and Object Dependency Graphs [74] both use objects to represent structure changes in their inter-view propagation mechanisms. However, these objects are only used to reconcile viewing object structures to their models, and to implement undo/redo and hierarchical change propagation mechanisms. They do not support presentation or monitoring of longer-term inconsistencies. The View Mapping Language (VML) [2], which primarily uses constraints to map changes between schema views, could be extended to provide some of these capabilities, with objects representing inter-view inconsistencies.

FIELD [63, 64], and its successors, such as DECFUSE [37], use selective broadcasting to propagate messages about tool events between multiple Unix tools. Limited forms of view consistency are supported by FIELD, and building such environments and integrating new tools into the environment requires much effort [53]. CSCW toolkits, such as Groupkit [66], which uses a similar approach to FIELD for informing multiple users' views of changes, also lack the range of inconsistency recording and presentation facilities required. CORBA-based multiple-view tools, such as those proposed by Emmerich [16, 18], may provide appropriate capabilities, by using combinations of remote object change broadcasting and shared data access for integrated tools. GTSL [17, 19] supports the generation of incremental consistency evaluation algorithms for tools by utilising document schemas and semantic relationship specification. GSTL also

supports versioned documents and concurrent tool interactions, allowing tools to support differings levels of granularity and user coupling for cooperative editing. While GSTL-generated environments, such as an integrated SEE for British Aerospace [5], support powerful cooperative work, configuration management and inter-document dependency management facilities, they lack the range of short- and long-term inconsistency presentation, monitoring and interaction capabilities of tools like SPE. A few systems directly 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 [21], which uses logic predicates to record inconsistencies between different viewpoints on software designs.

Some software development tools, such as Mjølner environments [49], take the approach of preventing tool users from representing software artefacts in multiple views at all, therefore theoretically limiting possible inconsistency management problems. However descendants of Mjølner-based systems for multiple-user software development [50] have introduced simple inconsistency presentation and highlighting techniques to support version control and multi-user editing. PECAN [61], Garden [62], and Dora [59] all support multiple views of software artefacts, kept consistent via MVC-style object observation or database triggers. While a rich range of views is supported by these environments, they provide limited support for managing long-term inconsistencies between views. They also rely on structure-oriented editing techniques in order to keep views consistent, which lack favour with software developers [4, 73]. The Cornell Program Synthesizer [65] uses constraint expressions to present semantic inconsistencies to developers in views of software artefacts, which can potentially exist for long periods, but these can not be interacted with nor be grouped.

Most CASE environments use the notion of a repository, with database view mechanisms to keep multiple views of artefacts structurally consistent, and triggers and queries to detect semantic inconsistencies [23]. For example, Software thru Pictures™ [72, 40], uses a Sybase database, EiffelCASE [41], uses persistent Eiffel objects, and Rational Rose [60], uses files. Inconsistencies detected by constraint triggers are acted on immediately and queried inconsistencies may lack information about the context (such as part of the software process) a change was made in, who made the change, when it was made, and why it was made. As inconsistencies detected by triggers or querying are generally not represented and stored in the tool repository, inconsistency representations can not be annotated with additional information about them at the time they are generated and stored for monitoring.

CASE tools generally use reverse engineering and merging tools to reconcile modified CASE and code views [41, 40]. They often require separate configuration management and shared repository tools to enable collaborative development. For example, Rational Rose uses a separate product, ClearCASE, to maintain shared CASE documents via a configuration management tool with checkin/checkout policies [60]. A similar approach is taken in EiffelCASE [41,] and Software thru Pictures [40]. This approach delays the detection of inconsistencies between design and code views and thus fails to provide immediate feedback to developers when such inconsistencies occur. This often leads to many inconsistencies between design and implementation views of a system that require a large effort to resolve later. Object Team [13] utilises a repository with built-in configuration management, continually synchronising design and code artefacts. This approach leads to intolerance of partial inconsistency between developers, which can greatly hinder useful parallel design and development exploration [21]. The increasing need for a range of multi-user support facilities in CASE tools [45] also means that architectural approaches which better facilitate the management of multi-tool and multi-person inconsistencies, such as selective broadcasting, CORBA-style remote notification and process-centred support, need to be used.

Meta-CASE tools usually provide some degree of support for generating multiple-view supporting CASE tools. MetaEDIT+ [43] and MultiView [1] provide limited view consistency management, based on database views. MetaView [67] provides view inconsistency management using constraints, with basic facilities to allow developers to monitor, group or interact with inconsistencies. They do not support the annotation of inconsistency information with context that can aid the user in understanding the reasons for a change. KOGGE [15] generated tools use a database with active object views to keep multiple viewpoints consistent at all times, with no tolerence for inconsistency.

Process-centred environments allow software developers to plan and coordinate their use of multiple tools and work on multi-user projects. Oz [11] and Merlin [57] use rule-based approaches to describing process models, and Oz utilises "enveloping" to integrate other tools with the process-centred environment [70], but do not provide management of inconsistency representations generated by integrated tools [51]. TeamWARE Flow [69], Action Workflow [52] and Regatta [68] use more developer-accessible, graphical process modelling languages, and provide simple interfaces to coordinating third-party software development tools, but do not manage inconsistencies. SPADE [8], ProcessWEAVER [20] and ADELE/TEMPO [10] provide more sophisticated facilities for integrating third-party tools, with some inconsistency management, event handling and constraints applicable to integrated tools. However, the degree of integration means only basic inconsistency monitoring can be facilitated, and inconsistency presentation and interaction within integrated tools is not supported [9]. The process modelling and enactment approach of [47] has been used to coordinate multiple view usage [58], allowing developers to specify how inconsistencies detected in multiple view tools can be handled.

4. Inconsistency Representation using CPRGs

The first four inconsistency management requirements we identified in Section 2.2. relate to the ability of software development tools to adequately represent software specifications and to detect and represent inconsistencies. As existing software architectures do not support all of these inconsistency management requirements, we have developed a new software architecture, Change Propagation and Response Graphs (CPRGs), for managing inconsistencies in multiple view software development tools [29]. We first motivate the need for this architecture and explain its design rationale, with a simple example of its use for inconsistency management. We then describe the realisation of this architecture in tools which support the construction of CPRG-based environments.

4.1. A Model for Inconsistency Management

Software specifications consist of a wide variety of "software artefacts" i.e. different kinds of information about software which together comprise a full (or partial) system specification at various levels of abstraction. Some of these artefacts are structured, textual or graphical documents, while others are more loosely structured. Graph-based structures tend to suit the representation of fine-grained software artefacts, such as abstract syntax trees and graphs for diagrams and code [4, 65, 7]. However, in order to produce efficient environments which can make use of existing tools, such as text editors, coarse-grained representations of parts of software specifications, such as the code associated with a class method, can also be represented as a single "artefact" [27, 59].

The CPRG software architecture is based on a graph-based representation for software artefacts, with attributed *components* linked by inter-component *relationships*. CPRG components can represent small, fine-grained software artefacts, such as classes, attributes, methods, etc. They can also represent coarser-grained artefacts, such as entire class interfaces, third-party tool documents, such as MS Word™ files, and interfaces to third-party tools and databases. Multiple views of software specifications are built with CPRGs using "view components", with these view components related to "base" (i.e. repository) components via "view relationships". This is illustrated in Figure 2 with a class method shown in two different views (one a graphical design view, the other a textual code view). The top windows are graphical and textual views with which the user interacts. The middle layer is a CPRG representation of these views, and the bottom layer is the repository of the environment containing a CPRG describing all software specification information.

Changes to graph structures representing parts of a software specification lead to inconsistencies when: i) related structures that share updated information are not changed, e.g. multiple views of a changed component are not appropriately updated; ii) semantic constraints on components are violated e.g. a class trying to use a method in another class uses the wrong number or type of arguments; and iii) specifications updated by one developer become inconsistent with those of other developers. CPRG components and relationships are used to embody both the structure of software specifications and the semantic constraints within and between parts of specifications. When these detect that related

components have changed, they carry out structural or constraint checks for inconsistency. Some CPRG components may embody only structure or only semantic constraint information, but often many embody both. When developing CPRGs, we chose not to introduce specialised kinds of relationships for e.g. structural relationships versus semantic constraints, in order to keep the model simple but also to ensure a homogeneous approach to handling both structural and semantic inconsistencies.

The state of a CPRG component is modified by an *operation*. When an operation changes the state of a component, objects describing this state change, called *change descriptions*, are generated. Change descriptions generated by a component are propagated to all conected CPRG relationships, which then decide to forward the change to other components, act on the change, or ignore it. If related structures are not updated to reflect the changed state of the modified component, change descriptions indicate structural inconsistencies. Change descriptions are also used to represent semantic inconsistencies, by having change descriptions generated when constraint violations are detected. Components generating or receiving change descriptions can also annotate them with extra information, such as the time and date the inconsistency occurred, the developer who caused it, what process model stage was enacted when it occured, the relative importance of the inconsistency, or any other additional reason for this inconsistency being represented. Change descriptions can be grouped with components to "record" the presence of inconsistencies associated with these components. These groups of change descriptions can be browsed by users and searched and acted on by software development tools. For example, the history components in Figure 2 are used to record the modification history of views and repository-level classes.

An additional reason for using change description objects, rather than constraints or traditional model-view change notification, is the homogeneous solution change descriptions provide for inconsistency management, tracking and versioning component changes, and collaboration support [29, 30]. Change description storage supports the formation of "modification histories" which track changes that components have undergone. In the Serendipity process modelling environment we also associate groups of change descriptions from software artefacts with enacted process stages, showing the history of work done for each stage when it was enacted [34]. CPRG change descriptions representing state changes can also be reversed to "undo" these state changes, or re-applied to repeat the state changes. This combination of recording changes in groups and undo/redo supports "deltas" for each component, supporting a basic versioning mechanism. Change description objects can also be serialised and broadcast between users' CPRG-based environments, facilitating a range of collaborative work facilities [31].

Figure 2 illustrates an example use of CPRG change descriptions to support inconsistency management in SPE. The name of a method in a class diagram is modified by the user (1), resulting in a change description being generated. This change description is propagated to the view and stored in the view's modification history, and propagated to the method view components's view relationship, which automatically updates the repository method component (2) to remove the structural inconsistency between repository and view. A change description resulting from this update of the base method component is generated (3), and then propagated to all related components interested in changes to the base method (4). If guiding software development in SPE with a Serendipity process model, the change description will also be annotated by Serendipity with information about the developer and enacted process stage. The view relationship informs all other views of this base component of its state change (5), and views are either fully or partially updated or inconsistencies recorded and presented to the user, with affected view components highlighted. Classes which use this method check semantic constraints on the usage (6). If the change to the method causes their use of it to become invalid e.g. their code calls the wrong name, or wrong arguments, a semantic change description is generated. If the user has specified this kind of inconsistency is important, the change description will be annotated with e.g. a "medium" importance indicating the developer must correct the change before the system can be compiled, or a "high" importance indicating the inconsistency should be resolved very soon. The owning class of the updated method is informed of its state change, and records the change description in its modification history (7), tracking all changes to the class. The original change description generated by the view component update can be broadcast to other developers sharing the edited view, to support a variety of collaborative editing and version merging facilities.

Figure 2. An example of inconsistency management using CPRGs.

4.2. A Software Architecture and Support Tools for Realising the CPRG Model

We have developed object-oriented class frameworks and an environment generator to allow software tool developers to use the CPRG architecture to develop multi-view, multi-user environments. We have used these frameworks and generator to construct several exemplar development tools and environments exhibiting a range of inconsistency management facilities.

4.2.1. MViews and Serendipity

Our first realisation of CPRGs was the MViews class framework for building multi-view editing environments [30]. We chose an OO framework approach to enable tool developers to reuse the basic CPRG functionality via inheritance and composition of CPRG-implementing classes, and then extend this basic functionality for use in their own tools by writing additional, application-specific code. We built a range of specialised classes for MViews that extend the basic CPRG model. These provide abstractions for building multiple view representations of software artefacts, a variety of inter-component relationships, a variety of graphical and textual editor building-blocks, and support for collaborative work in MViews-based tools.

We implemented MViews in Snart, an object-oriented Prolog extension [26]. We chose to represent change descriptions in our Snart implementation of MViews as Prolog terms, rather than Snart objects, for speed of generation, propagation and storage. Change description terms in our MViews implementation represent CPRG component state changes and constraint violations, with a Prolog list appended to enable the annotation of change descriptions with additional information about inconsistencies. Collaborative view editing and version sharing is supported by broadcasting serialised terms between users' environments and annotating change descriptions with user names. 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.

In MViews-based environments, such as SPE, it became apparent that users require additional "work context" information about inconsistencies, and for inconsistencies to be grouped not only by affected artefacts but also by user and the context in which the inconsistency arose. We chose to "capture" additional reasons about inconsistencies (and software artefact modifications in general) by using software process model information to annotate MViews change descriptions with user, time and date, process model stage, and reason for stage enactment information. This led to the development of the Serendipity process modelling environment, and integration of Serendipity and MViews-based environments [31, 34]. We modified MViews so that any change descriptions generated by view components or base layer components are forwarded to Serendipity (if in use), and these change descriptions are annotated with enacted process model information. They are also copied and stored against the enacted process model stage to facilitate grouping of inconsistency representations and modification histories with process model stages in Serendipity. Figure 1 shows examples of annotated change descriptions in SPE, and also stored change descriptions originating from SPE modifications stored against a Serendipity process model stage.

4.2.2. JViews, JComposer and Serendipity-II

We built many multiple-view, multi-user software development environments with MViews. However, MViews-based environments suffer from slow performance due to the Prolog implementation of MViews, difficulties in integrating third-party tools not built with MViews, and a large amount of effort required by developers to implement MViews-based tools. This led to the development of JViews, a Java-based implementation of CPRGs, which uses and extends the Java Beans change notification mechanism to represent change descriptions and support change description propagation [33]. JViews is an object-oriented class framework with similar capabilities to MViews, but its architecture is much more open for tool development and integration. JViews uses the Java Beans componentware API to allow events from third party tools to be represented and used within JViews environments, and to be able to send instructions to third party tools which provide component-based interfaces. A variety of components supporting collaborative view editing, component persistency and repository management, and software artefact representation are provided by JViews. JViews does not currently use a common architecture for distributed object management, such as CORBA. However, we are investigating use of the remote object change notification mechanism in CORBA for representing change descriptions and their propagation for JViews-based environments.

To reduce development effort with JViews, one of our first JViews-based environments was JComposer, a tool for specifying CPRG-based environments and generating JViews-based implementations [33, 35]. We decided to generate JViews implementations of tools rather than interpret JComposer specifications to ensure efficient, stand-alone and interoperable tools would result. Figure 3 shows an example of part of the specification of the Serendipity-II process modelling tool, a reimplementation of Serendipity using JComposer and JViews. The JComposer view on the left shows part of a specification (which uses the CPRG notation) being developed. A view from the generated environment running is shown on the right. Serendipity-II can be used to provide a context for inconsistencies in JViews-based environments, in the same manner as Serendipity does for MViews-based environments.
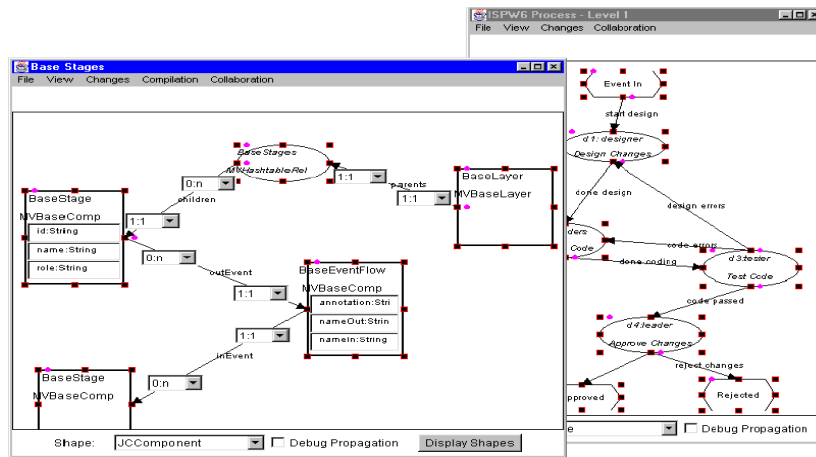
Figure 3. An example of the JComposer tool and the generated Serendipity-II in use.

5. Querying and Presenting Inconsistencies

Any environment supporting multiple views needs to either automatically resolve view inconsistencies or to inform developers of unresolved inconsistencies [53, 61, 63]. Often inconsistencies between multiple views can be automatically resolved by modifying the state of one or more views to reconcile them to the modified state of other views. An initial consideration is whether a change description can and should be automatically applied to a view receiving the change to rectify the inconsistency it represents. An example where this is possible is the renaming of a class in an SPE class diagram, which is easy to propagate to and automatically effect in other class diagrams showing the class. CPRG-based environments readily support this behaviour by having response methods in the components that receive change descriptions pattern match against received changes and perform actions to rectify inconsistencies. SPE uses this technique extensively for maintaining graphical view consistency. Many existing multiple-view software development systems support only this level of view consistency management. If a change can be directly made to another affected view, it is performed by the environment. If it cannot, it is ignored and the user of the environment is usually never informed of a potential inconsistency.

Just because an inconsistency can be automatically rectified it is not necessarily the case that it should be. For example, a far-reaching change made by an inexperienced colleague may be better handled as an inconsistency that is presented to other developers. In addition it is often important to track that an inconsistency occured and that it was automatically resolved by the environment. Thus in SPE we have chosen to have all changes to a view or a component recorded in modification history lists, able to be reviewed using the techniques described in the following sections. For any important changes which are automatically resolved by an environment, it is also often useful to highlight the change *after* it has been made, to draw developers' attention to it. For example, in SPE changes such as renaming classes and methods may be partially automatically resolved between views, but the changed items in views are highlighted.

5.1.   Highlighting the Presence of Inconsistencies

CPRG-based environments inform components of the possible presence of inconsistencies by propagating change descriptions to them, and record the presence of such inconsistencies by associating change descriptions with them. These environments must then inform developers that such inconsistencies have been detected, either in a "context-dependent way" (e.g. indicating what they affect in the views with which developers are interacting), or in a "context-independent way" (e.g. by grouping and presenting related inconsistencies together). Highlighting parts of a view supports a basic "context-dependent" approach to informing developers of inconsistencies which affect the view.

A variety of techniques can be used to highlight the presence of inconsistencies, including shading and colouring graphical icons, changing text font characteristics, or blinking affected icons. Annotating the name of a view's window, or the values of text displayed in the view are also often appropriate and easily

implemented. Many of these techniques are equally applicable to textual and graphical views, and many can be usefully combined. The choice of technique often depends on how developers can most appropriately be informed of the presence of different kinds of inconsistencies. For example, inconsistencies which need quick resolution to allow software development to proceed should be immediately presented so that developers readily notice and act upon them. Inconsistencies which can be tolerated for longer periods, or which do not have a large impact on the information displayed in a view, are usually more effectively presented in less dramatic ways, or may even be highlighted only when requested by developers.

As a simple example, consider the screen dump from the dialogue definer of SPE in Figure 4. This shows three views of a dialogue under design, a graphical drag-and-drop composition (dialog1-dialogue), a textual specification (dialog1-Dialog Predicate) and an example of the running dialogue. In this example, the 'Ok' control button in the graphical view has been shifted, resulting in a semantic constraint that dialogue components not overlap the dialogue border being violated. A change description has been generated representing this inconsistency and associated with the dialogue view's Ok button component. Such an inconsistency must be resolved before the dialogue specification can be used. Thus SPE immediately indicates the presence of such semantic inconsistencies by shading the Ok button icon, to highlight it. This technique could be used in combination with other context-dependent inconsistency presentation approaches, such as boldening the textual Ok button specification in the right-hand view, renaming the views to e.g. "dialog1-dialogue (error)" to indicate the view has an inconsistency, and so on.

Figure 4. Indicating inconsistencies in graphical views.

5.2.   Textual Presentation of Change Descriptions

While the presence of inconsistencies can be indicated in a variety of ways, developers often need more information about them. For example, a developer may not understand why the Ok button in Figure 4 has been highlighted and wants to view more information about the inconsistency this indicates. As described in Section 4, change descriptions may embody a wide variety of information about an inconsistency. Textual forms of change descriptions can be presented to developers by inserting them into views or displaying them in pop-up menus and dialogue boxes. An important characteristic of all CPRG change descriptions is that the inconsistency information they embody can be "unparsed" and viewed in a textual, human-readable form.

For example, Figure 5 shows the textual forms of several change descriptions from SPE indicating: 1) the addition of a method to a class in SPE; 2) a semantic error denoting an unknown method is being called; and 3) an annotated analysis change. For change description #1, additional information from Serendipity

has been used to annotate this particular change description, i.e. the process stage (`aff.2.1:Design Changes`) and developer making the change (`judy`). This annotation allows users to determine the reasons the change was made or inconsistency detected, who caused the inconsistency, and so on. For change description #2, the "importance" of the semantic inconsistency is indicated with the three stars prefixing the description. For change #3, an additional annotation indicating the change was made in an analysis diagram has been added, indicating that when this change description presentation is viewed for design and/or code views, the change needs to be appropriately propagated to specifications at these levels of abstraction.

```
1. [aff.2.1:Design Changes-judy] add feature customer::calc_fines
2. *** Unknown method 'check_limit' being called in customer::check_cr_limit
3. [*** analysis] change association [customer::acc-of, ->, account] arity
from  1:1 to 1:n
```

Figure 5. Textual form of a change description

5.3.   Grouping of and Querying for Inconsistencies

Change descriptions can be grouped and associated with various CPRG components they affect. Such grouped change descriptions provide an underlying "database" for querying about inconsistencies. By applying selection operations on this database, collections of change descriptions relating to a particular component, view, process stage, or inconsistency type can be constructed, and then appropriately presented to the user.

For example, SPE keeps histories of modifications and semantic constraint violations for all class components and views, but also records groups of all constraint violations, unresolved "inter-specification" changes (e.g. design changes affecting code and vice-versa), inconsistencies resulting from version merging, and changes by Serendipity process stage. These groups of stored changes are then used to create a variety of change and inconsistency presentation lists for users to interact with. Figure 1 shows two such lists, one for a Serendipity process stage, and one for an SPE artefact (class customer). Each change description in the lists is displayed in textual form, annotated by a sequence number indicating the order they were generated. Developers can also configure environments constructed with our CPRG-based tools to record inconsistencies in user-specified ways (see Section 8).

An example use of presenting grouped inconsistencies is shown in Figure 6. Changes made to an SPE class are shown, with some changes having been translated from changes made to a same-named EER entity [71]. Items in this change history list 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 i.e. an inter-notation translation inconsistency occured. For example, adding an 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 could not automatically be done, as the EER notation does not support the distinction between different kinds of relationships. This technique of presenting inconsistencies between design notations in a textual list is combined in SPE with highlighting the presence of possible inter-notation inconsistencies by shading icons in a view affected by changes to another notation view [71].

```
        *
        *

        *
        *
        *
        *

        *

        *
        *
```

Figure 6.   Graphical view inconsistencies.

5.4.   Displaying Inconsistency Representations in Views

One application of grouped change descriptions commonly used by our tools is to present currently outstanding inconsistencies by annotating the contents of views. That is, rather than just highlighting view components to indicate the presence of inconsistencies, inconsistency descriptions are inserted into the view. This provides more immediate information to the user than simply indicating which parts of a view are inconsistent. We achieve this either by inserting textual descriptions of change descriptions directly into a textual view or by adding pop-up menu items to a view where such textual descriptions can be readily accessed. Care must be taken to ensure these descriptions are put into a logical place for developers to access, and to distinguish them from actual software artefact specifications. We have found specially distinguished comment areas for textual specifications and pop-up menu handles for graphical views most appropriate.

For example, Figure 7 shows an SPE class interface view with several change descriptions representations inserted into a special header region at the top of the view. These describe inconsistencies between this view and other, modified views of the program, and also semantic errors caused by constraint violations (e.g. same-named items). In this example, change 32 indicates the address attribute has been renamed to address1, change 33 indicates addition of attribute address2, change 36 indicates addition of a client/supplier relationship between the customer::add_invoice method and the invoice::create method, and change 44 indicates a compilation (semantic) error due to the duplicate calc_total_owed methods. We chose to present change descriptions at the begining of SPE class and method implementation specifications, and when developers open such views these inconsistency presentation regions are updated with new inconsistencies inserted after older ones. We also allow developers to edit this text to remove inconsistencies they have actioned or do not want to continue to see in the view. However, such inconsistencies are kept and can still be queried for and viewed in a dialogue.

The first three of the presented inconsistencies in Figure 7 might have been made in a graphical view and thus the change descriptions inform the programmer of (possible) view inconsistencies between this

textual view and the modified graphical view. Changes 32 and 33 can be automatically applied by SPE to the textual view. The developer can configure SPE to always automatically update the view's text to reflect such changes, rather than displaying the change descriptions. To resolve change 36, the programmer may, at some later time, modify the customer::add_invoice method to insert an appropriate method call and arguments, and possibly update the customer class so that an appropriate reference to invoice class objects exists. It can be difficult to describe the full cause of semantic inconsistencies, such as Change 44. Often an environment may only inform users of the editing change(s) that caused the inconsistency and the semantic constraint violations detected.
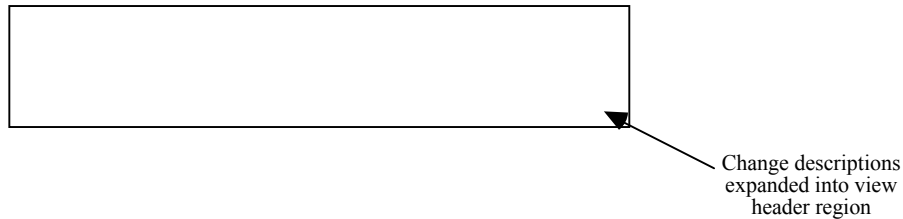

Figure 7. Indicating inconsistencies in textual views.

An interesting use of the above technique in SPE is its application to documentation views. SPE allows artefacts, such as classes, to have associated documentation views describing them. Change descriptions affecting the artefact are forwarded to the documentation views and automatically inserted into the view header, acting as an indicator that the documentation may require updating to reflect the change.

6. Interaction with and Resolution of Inconsistencies

Once developers are aware of the presence of inconsistencies, they may wish to interact with inconsistency presentations or indeed resolve these inconsistencies. Interaction may simply be clicking on an inconsistency presentation to gain more information about it. For example, clicking on the shaded Ok button from Figure 4 to be shown a detailed description of the inconsistency, like those in Figure 5, in a dialogue. Interaction may also include selecting inconsistency presentations and requesting the environment to: resolve them automatically; mark them as "seen"; change the importance or other information associated with them; move them to a history list for later action; or delete them (i.e. the inconsistencies are tolerated). Because inconsistency presentations in our systems are representations of CPRG change description objects, providing developers with the ability to interact with inconsistency presentations can be seen as providing facilities to manipulate these change descriptions. This section illustrates some of these techniques as used in SPE, and the rationale for their choice in different situations where developers need to interact with inconsistency presentations.

6.1. Selection and Resolution of Inconsistencies

Many structural inconsistencies between views can be automatically resolved by our environments. For example, for textual and graphical views SPE can automatically expand added or hide deleted classes, methods, method arguments, local variables and attributes, and can automatically rename classes, methods and attributes, and change attribute and method argument names and types. In graphical views, SPE can automatically expand added, change modified and hide deleted classes, class components and inter-class analysis and design relationships [27]. When structural inconsistency resolution actions are automatically carried out, changed view items are highlighted to indicate they have been modified and the actioned change descriptions stored in a history list associated with the view for developer perusal. When such structural changes cannot be automatically resolved the inconsistencies are presented as described in Section 5.

Developers may choose to have inconsistency presentations shown in a view or associated history list dialogue, and not be automatically resolved by an environment. For example, with SPE we have found developers want automatic resolution when using graphical views, but with textual views often want all inconsistencies shown in the textual view header, whether they can be automatically resolved by SPE or not. This is because even when apparently simple graphical view analysis and design changes are made, such as renaming attributes etc. which can be automatically applied to some affected textual implementation code, developers like to be informed of such inconsistencies and determine when they are resolved. They also like to have both automatically resolveable and non-automatically resolveable inconsistencies handled in the same manner for textual implementation views, but prefer automatically resolveable analysis and design changes to be applied to graphical views when they are detected by SPE. This behaviour can be changed by developers if required, as described in Section 8.

For inconsistencies that can be automatically resolved, the user can select one or more inconsistency presentations in a dialogue or view, issue a request to implement the change, typically via a pull-down or pop-up menu item, and are informed of the results. Figure 8 shows an example of such developer-requested automatic view inconsistency resolution in SPE. The user has selected all of the change descriptions in the view header and asked SPE to update the view's text. The first two changes can be automatically applied by the environment and result in attribute address being renamed to address1, and addition of attribute address2. Both change descriptions are deleted indicating successful update of the view. The other two changes, however, cannot be automatically applied, and the change descriptions are left in the view's text to indicate this. Updates selected in a dialogue but which could not be successfully applied by the environment are highlighted or shown to the user in another dialogue.

selected
change descriptions

change descriptions
deleted from view header
by environment

view text updated
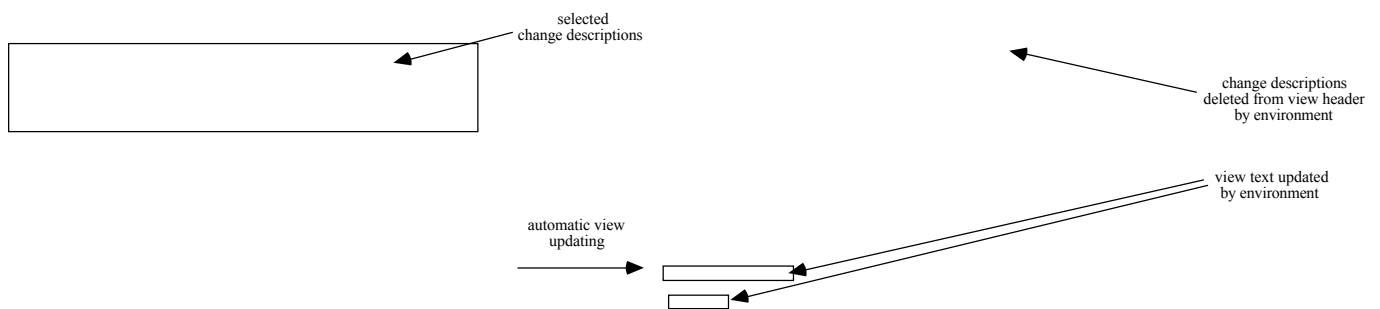by environment

automatic view
updating

Figure 8. Automatic resolution of view inconsistency under developer control.

6.2. Selection of Optional Update

Often environments can partially resolve an inconsistency, or may determine several possible alternative approaches to resolving it. Environments in which such partial automatic inconsistency resolution is possible include those which integrate multiple design notations, such as OO, ER and NIAM models [71], multiple levels of specifications, as in SPE [27], and implementation, documentation and formal specifications [30]. Often a change made to one notation view can be partially translated into a change in views using another notation. The user may be able to complete the translation by choosing from a range of possible view changes.

An environment can assist in determining a partial inconsistency resolution or a range of alternative strategies. The environment can then make the partial change and leave the developer to complete the resolution, or facilitate the developer in choosing their preferred inconsistency resolution strategy. Developers should be informed that a partial change has been made or that they need to complete a change, using similar highlighting and presentation techniques to those described in Section 5. Developers can choose an appropriate resolution strategy, for example, via a pop-up menu with resolution choices or by having several alternative change descriptions presented which describe each resolution and allowing developers to select the one they want.

Consider SPE facilitating the translation of changes between OOA and EER notations when a relationship is added in an EER view. For example, an EER relationship is added between customer and invoice entities, and SPE translates this into the addition of a relationship between customer and invoice classes in OOA/D views. However, more information 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 an OOD client/supplier relationship, it may specify a method call between objects and thus caller/called method names and arguments must be specified. As this information is not specified in EER views, we chose to have SPE default the relationship to an OOA association relationship, and let the user refine this further by changing it to an aggregation or client/supplier relationship by adding extra information. SPE indicates the new relationship has been defaulted by highlighting it, and provides a pop-up menu to allow the user to easily change its type and add appropriate extra information. 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.

6.3. Manual Resolution of Inconsistency

Some inconsistencies between views can be presented to developers but not resolved in any way by an environment, although the environment may suggest different ways the developer might manually resolve them. In this situation, the developer must manually resolve the inconsistency (which can be quite difficult and time-consuming). Using different highlighting techniques, our environments inform developers of the presence of such inconsistencies and may indicate they can not be resolved by the environment. Developers then resolve the inconsistency manually, indicating they have resolved it by deleting the inconsistency presentation, moving it to another history list, or associating information with it indicating its resolution (or partial resolution).

For example, the addition of a design-level client/supplier relationship in SPE is usually implemented as a code-level method call. Such a change cannot 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. Similarly, semantic errors such as duplicate method or attribute names, type mis-matches and the calling of non-existent methods in other classes all require developers to take some appropriate action to correct the problem. Figure 10 shows an example of this manual inconsistency resolution to resolve a simple semantic inconsistency in a class definition view.
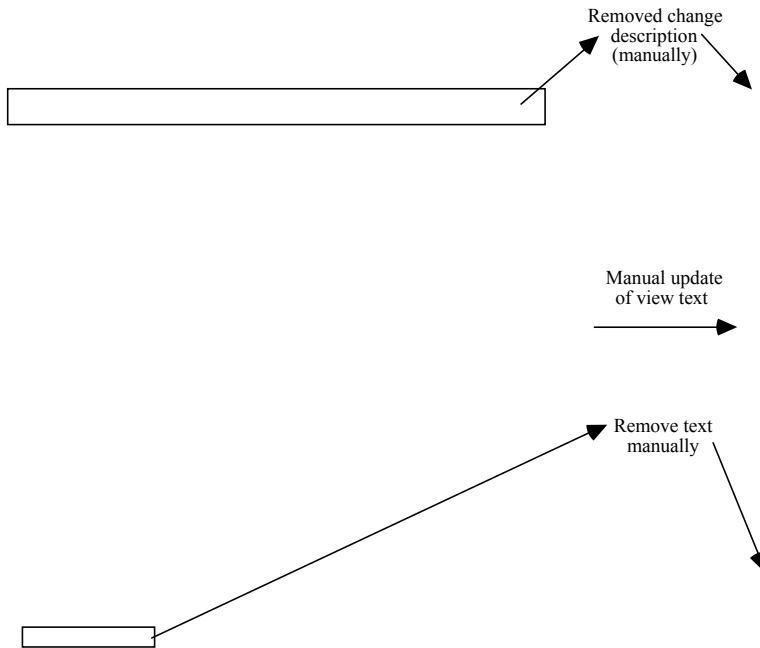
Removed change
description
(manually)

Manual update
of view text

Remove text
manually

Figure 10. Manual resolution of inconsistency.

6.4.   Other Interactions with Inconsistencies

In addition to resolving inconsistencies, users can interact with inconsistency representations in other ways. An environment can add interaction "hot-spots" to highlighted inconsistencies, providing access to a more complete decription of the change in a dialogue or pop-up menu, or display the view whose update caused the inconsistency. Developers can also be provided with a mechanism to annotate an inconsistency themselves, to move it or copy it to other inconsistency recording groups, or to delete it (i.e. tolerate the inconsistency). We have found that providing inconsistency presentation selection facilities in views and dialogues, or even pop-up menus associated with inconsistency presentations, effectively supports such developer and inconsistency interactions.

For example, Figure 11 shows an SPE change history where the developer can select inconsistencies and ask for their affect to be undone/repeated (if structural changes), ask for more detailed information to be displayed, annotate the inconsistency, or delete it from the list. Developers may even create their own "user defined" inconsistencies and add them to the list. These do not necessarily represent a view inconsistency, but rather serve as user-defined documentation, perhaps describing the effect of a group of lower level changes which have been made. Such user defined change descriptions are associated with specific software components, and, like change descriptions representing structural and semantic inconsistencies are propagated to other views of these components. They also serve a useful purpose in supporting context-dependent communication in cooperative environments, allowing cooperating users to interact by "messages" sent via the change description mechanism [27, 29].

Figure 11. Example of a user-defined change description from SPE.

7. Handling Inconsistencies During Collaborative Software Development

Computer-Supported Cooperative Work (CSCW) systems may use a multi-view editing approach to sharing and modifying information [38, 53, 59]. Inconsistencies between views are often difficult to resolve, as developers may share and modify different versions of the same view in incompatible ways, and may concurrently modify views at different levels of abstractions. We have used the techniques described in Sections 3 to 5 to support a range of inconsistency management techniques for collaborative software development [30].

7.1. Inconsistencies During View Version Merging

CPRG-based environments support the creation and sharing of multiple alternate versions of views i.e. copies of a view which can be asynchronously modified by developers and then merged to produce a consistent, shared software specification. While alternate versions of a view are being modified independently, inconsistencies between these views may easily result. For example, a class method modified by one developer may be deleted or renamed by another. We use the CPRG change description storage mechanism to construct "deltas" between view versions, i.e. sequences of changes made by developers to different alternate versions of a view. Developers may then combine these sequences of view modifications by actioning each change in turn on the orginal version of a view. The non-sequential undo/redo facility supported by CPRG change descriptions is used to support this. Structural inconsistencies may occur when some of the modifications made by one developer are incompatible with those of another. Similarly, semantic constraint violations may be present in the alternate view versions being merged, or may be present in the new version resulting from the merging process. Both structural and semantic constraints resulting from version merging can be presented to developers using the

techniques of Section 5, and developers may resolve these inconsistencies using the techniques of Section 6.

An example from SPE is shown in Figure 12, where two developers have independently modified two alternate versions of an SPE design diagram, and these changes have been merged to produce a new version of the view which tries to incorporate all of these changes. The developer doing the merging repeatedly selects some or all of the change description presentations for each view, and asks for them to be actioned on the new version being created (using the "Redo" button). In this example, two structural inconsistencies and a semantic inconsistency have been detected during this process, and these inconsistencies have been presented to the developer in the bottom dialogue.

```
           (V1.1a)                              (V1.1b)
                          (merge)
 structural                ↘  ↙
 conflict                 (V2.0)


       ┌──────→
       │
       ┌──────→
       │
 semantic
 conflict
```
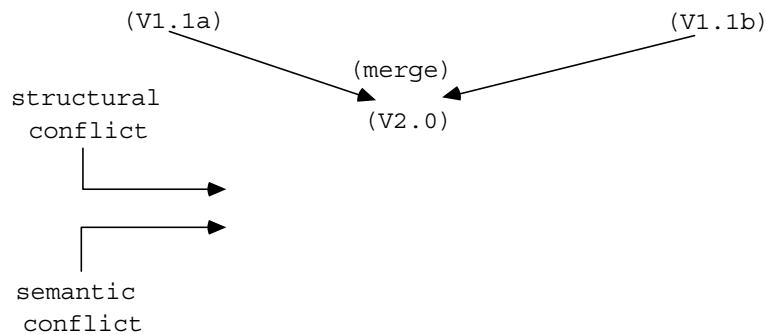
Figure 12.   Version merging and presentation of merge conflicts via change descriptions.

7.2. Inconsistencies During Synchronous View Editing

Often developers wish to synchronously or semi-synchronously edit software specification views, particularly those at a high level of abstraction such as analysis and design views. We can use the inconsistency presentation and interaction techniques of Sections 4 and 5 to allow developers to see and act upon the presence of inconsistencies introduced by another developer editing a shared view or a related view. Inconsistencies can be annotated with the name of the developer causing the inconsistency, and presented to users in views, pop-up menus and dialogues as appropriate. Automatically resolved inconsistencies can be stored for perusal by other developers, and automatically modified view items highlighted. We have found assigning colours to each developer to indicate who has last changed what to be useful. Inconsistencies which can not be automatically resolved are presented to developers, and an appropriate resolution strategy negotiated using annotation of change descriptions, textual chats and/or audio conferencing.

Figure 13 shows an example of semi-synchronous editing in SPE utilising inconsistency presentation and interaction techniques. When user "rick" edits his version of the OOA view at the top, changes are propagated to user "john's" environment (a screen dump from which is in Figure 13). Inconsistencies introduced by Rick's editing are shown in dialogues and in textual views. John can configure his

environment to automatically apply structural inconsistencies to his views. If desired, he could select inconsistency presentations and ask SPE to apply them (if possible), or could negotiate about the inconsistencies presented with Rick. Synchronous collaboration is also supported by SPE; whenever one user changes a shared view, other users synchronously editing see the view updated. Collaborating developers do not get to judge whether or not they want inconsistencies resolved in different ways with synchronous editing, and need to negotiate closely with collaborators to ensure all agree on changes made and any inconsistency resolution techniques used. As SPE associates groups of all inconsistencies and changes made with corresponding views and repository components, developers can peruse these histories when necessary, and interact with them to reverse or repeat historical modifications and to track long-term inconsistencies.
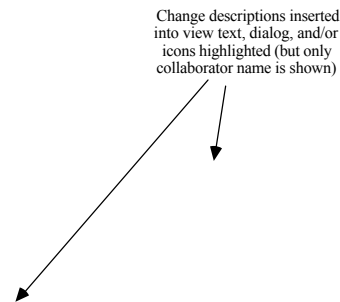
Change descriptions inserted into view text, dialog, and/or icons highlighted (but only collaborator name is shown)

Figure 13.   Semisynchronous view editing via change description broadcasting.

7.3.   Process-guided Inconsistency Management

Version editing and merging supports quite loose collaborative software development, whereas synchronous and semi-synchronous editing support more tightly-coupled work. We have found that additional process support for group work is also an advantage, enabling inconsistencies generated by multiple developers to be annotated with process stage information and to be grouped by process stage and/or developer causing the inconsistencies. This annotation and grouping of inconsistencies by process stage allows developers to more effectively manage a large number of inconsistencies between multiple views of software specifications. We found that simply grouping inconsistencies by software component or view alone did not scale up for multiple developers working on large projects, as developers often had insufficient information about why inconsistencies had occurred. They also want to view inconsistencies indexed not solely by view or software component, but by the same task in which they are introduced (i.e. those occuring during the same enacted software process model stage).

Our environments support the guidance of software development by the use of Serendipity process models. Serendipity, if in use, is sent change descriptions by environments like SPE, and annotates these change descriptions with information about the enacted software process stage for the developer causing inconsistencies to arise. These inconsistency representations are also grouped with the enacted process stages to give an orthogonal view of work to the standard CPRG component- and view-centred model.

8. Configuring Inconsistency Management Behaviour

While inconsistency management strategies as outlined in Sections 4 to 6 may be hard-coded into CPRG-based environments, they may not always match the tool users' requirements. This is especially true when multiple developers are involved on a complex project and the way different kinds of inconsistencies are

managed evolves over time. Environments handling inconsistencies in inappropriate ways may hinder rather than assist software developers. Ideally therefore developers need ways of configuring the inconsistency management techniques they employ. Developers may wish to extend the inconsistency detection schemes used by an environment, indicating additional kinds of structural changes they want propagated between software specification views and specifying additional semantic constraints on software specifications. They may want to change the ways in which inconsistencies that have been detected are presented to them. Similarly, they may wish to have inconsistencies stored in specific places for later perusual, have inconsistencies resulting from other developers' work forwarded to them, or to have their environment carry out specified operations when specific kinds of inconsistencies are detected, to automatically resolve inconsistencies or manage them differently.

For example, when using SPE and Serendipity, structural and semantic inconsistencies are grouped by software component and enacted process model stages. However, developers may wish to group some inconsistencies in different ways. They may also wish to be informed or have a new inconsistency resolution strategy applied when specific kinds of inconsistencies are detected.

To support developer configuration of inconsistency management in Serendipity, we have developed VEPL, a special-purpose visual event handling language [34]. We have since generalised this language for use in all JComposer-generated systems [33]. The event handling language provides users of CPRG-based environments a flexible mechanism for extending the environment behaviour by: specifying new semantic constraints and structural change propagation mechanisms; detecting certain kinds of inconsistencies which should be acted upon; grouping and presenting inconsistencies in various ways; and automatically performing developer-specified actions when inconsistencies are detected. Inconsistency management strategies can be specified in JComposer using VEPL and thus generated environments have this behaviour hard-coded. Users can also utilise a form of VEPL at run-time, i.e. when using a generated environment, to dynamically reconfigure inconsistency management behaviour.

Figure 14 shows a simple example of a Serendipity visual event-handling model being used to detect and then act upon an inconsistency occuring in SPE. The rectangles are "filters" which take change descriptions and pass on those that match user-specified criteria. The ovals are "actions" which perform simple or complex operations upon receipt of change descriptions (usually from filters). In this example, the development team have determined that multiple inheritance should not be permitted for a particular design. When this VEPL specification detects a developer attempting to use this construct in an SPE view, the change is automatically aborted and the developer informed why this was done. The change is then automatically aborted and the developer informed why this was done. The simple conceptual nature of Serendipity's filter and action event-handling model was designed to allow end-users of our environments to easily extend their behaviour, rather than having to use complex, low-level textual event handling code.

process stage artefact

filters changes related
to video class artefact

filters Add Generalisation events

checks if more than
one generalisation for a class

informs user of
an error ("semantic
constraint" violated)

causes SPE to reverse
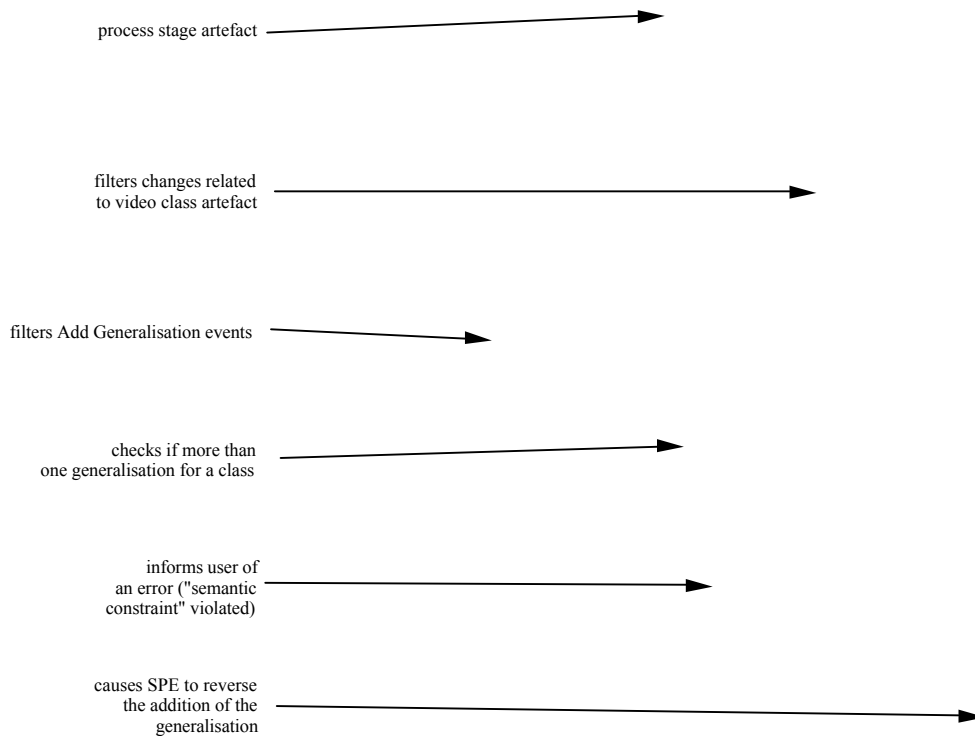the addition of the
generalisation

Figure 14. An example of constraining SPE via a Serendipity VEPL view.

The provision of these configuration facilities has interesting implications for collaborative software development. For example, two developers may be collaborating by sharing different versions of a view, but they may specify different filter and actions for the view i.e. different inconsistency management strategies. This may mean changes or inconsistencies made by one developer are not not detected and presented to the other developer. There are a variety of ways to address this problem. Developers can be informed when other developers change the inconsistency management behaviour of their environments, as the addition, deletion or modification of filters and actions are themselves documented by change descriptions. Alternatively, additional filters and actions can be specified by project leaders which constrain the degree of configuration allowed by members of a project team, by constraining the use of filters and actions. In SPE, we document the configuration of inconsistency management by recording filter and action component modifications in Serendipity, and leave it to the project team to negotiate about how inconsistency management configuration should be handled.

An additional benefit of filters and actions is that they support loose collaborative development without necessitating the use of defined software process models. For example, Serendipity can be used with SPE to specify inconsistency management strategies for multiple developers, but without using its software process modelling and enactment facilities. As a codified software process is not used, inconsistencies are not annotated with process stage information nor grouped by process stage. However, filters and actions could be defined to, for example, detect when another developer modifies an abstract specification and to inform other developers of this.

9. Experience and Evaluation

We have deployed the inconsistency management techniques described in the previous sections in the development of a wide range of software development tools. Some examples of these application domains include:
- object-oriented software development [27], including integrated analysis, design, code, Object-Z formal specification, and documentation views.

- collaborative software development tools, including asynchronous and synchronous editors [30] and process modelling and enactment [34]
- integrated EXPRESS-G graphical and EXPRESS textual views [3] for product modelling applications
- integrated OOA/D, EER, and NIAM graphical design tools, with textual relational database views [71], and method engineering facilities via process models and event handling configuration [32]
- integrated graphical and textual specification views for graphical user interface development [29]
- visual programming systems, including visual tool-abstraction [28], user interface [39], and programming systems [48].

While most of these systems have been built using the MViews framework, it takes considerable amount of effort for other developers to build MViews-based tools, due to the complexity of its OO framework. MViews-based tools are also difficult to integrate with third-party systems, and suffer from performance and "scaling-up" problems [30]. We have built JComposer itself, an ER modeller, and a Serendipity-II prototype using JComposer, which has significantly improved the ability of tool developers to quickly design and build tools which use our inconsistency management techniques. JComposer environments are also easier to integrate with third-party tools, as they have a more readily extensible Java Beans-based component implementation.

Our MViews-based environments have been deployed for use on small academic software development projects, and many have been deployed for larger student software development. For example, SPE has been used to facilitate multiple student OO software analysis, design and documentation using its integrated tools. Serendipity has been used by academics and students to describe and enact a wide variety of software and business process models. While MViews-implemented environments have provided inadequate performance and tool integration support for full-scale deployment, we are currently completing the JViews-implemented JComposer and Serendipity-II for use on a "real" multiple person software development project. Usability studies are to be conducted on these tools, in addition to file management, communication and visualisation tools we have been developing, to assess how well they contribute to inconsistency management for multiple person and multiple view software development.

Based on development and use of these environments, we have found that CPRG change descriptions provide an adequate representational mechanism for all inconsistencies we have needed to deal with in these problem domains. The CPRG change propagation and storage mechanisms, and the ability to embody structural and semantic inconsistency detection mechanisms in CPRG components and relationships, have also all proven to be suitable for the inconsistency management needs in our tools.

Users of our multiple-view editing tools like having view inconsistencies representations to see and interact with [27, 29], as these give users immediate feedback on the accuracy and consistency of their views. The presence of inconsistency presentations also assists users in determining where in the view inconsistencies exist and how to begin resolving them. For both system development and maintenance we have found the SPE-style approach of presenting and allowing interaction with inconsistencies in appropriate views to better support incremental design and code changes than separated reverse engineering and change merging tools. Environment implementers can choose between unobtrusive techniques to inform users of view inconsistencies, such as auto-expansion and highlighting, and presenting change description representations in dialogs immediately when a view is selected. We have found that it is often useful to present change descriptions, or highlight affected view components, even when inconsistencies have been automatically resolved, allowing developers to confirm a correct modification has been made. Environment implementers must, however, be careful not to 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 dialogue or displayed by interaction with the inconsistency presentations.

We have found that for inconsistency management techniques to be most effective, developers need to be provided with appropriate inconsistency interaction techniques. This is essential for structural inconsistency presentations that can be automatically resolved by the environment, and for inconsistencies which can be partially resolved or for which multiple resolution strategies exist. Developers should be able to easily select an appropriate automatic full or partial resolution strategy, otherwise they become frustrated at being presented with inconsistencies but without what they see as "obvious" resolution strategies. The ability to request information about inconsistency presentations, and to move between inconsistent views by selecting presentations, are important for developers to effectively navigate multiple, inconsistent views. Similarly, the ability to annotate inconsistency presentations, add user-defined "inconsistencies", and to move, delete or change the importance level of inconsistencies all foster a sense of inconsistencies which can be manipulated. Developers build a mental model of "interactable" inconsistencies in our environments due to the way inconsistencies are presented and able to be manipulated, so environments should whereever possible support this model.

We have found our presentation and interaction techniques have worked well for small environments with relatively simple views, but also scale up for larger environments with complex views and multiple users. Inconsistency presentation and interaction techniques help to foster effective synchronous and asynchronous collaborative development with multiple views. The use of Serendipity to annotate inconsistencies with process model stage information, and to group inconsistencies by stage rather than artefact, has proven a major enhancement over simply broadcasting user name-annotated inconsistencies.

Some of our earlier environments suffered from providing insufficiently powerful configuration facilities [27, 34]. Both Serendipity and JComposer provide visual languages to allow end users to configure their own inconsistency management behaviour. This allows software developers to specify the inconsistency detection, recording, presentation, monitoring and interaction techniques appropriate for different kinds of software artefacts and for different parts of their software process, alleviating many of the problems encountred in our environments without such facilities. Users of our environments have found the ability to tailor inconsistency management, even in basic ways, assists in making environments more effective [36].

10. Summary

Our experiences with handling inconsistency management in software development environments has indicated the need for:
- a flexible model for detecting, representing, storing, and propagating the wide range of inconsistencies that can occur, and which enables these inconsistency representations to be easily augmented with extra information, categorised and grouped as necessary
- a software architecture which efficiently realises this inconsistency management model, and support tools which assist in the construction of environments utilising this model
- a range of inconsistency querying, presentation and interaction techniques allowing users of these environments to ask for and/or see inconsistencies which affect views they are working with, and which enable them to monitor and/or resolve such inconsistencies in the most appropriate ways for the kind of view and the developer's preferences
- provision of multi-user capabilities allow inconsistencies which affect several developers to be managed, and configuration capabilities allowing developers and development teams to control inconsistency detection, storage, propagation, presentation and interaction/resolution.

We have developed a software architecture which represents inconsistencies as "change description" objects. Change descriptions are propagated between multiple-view representations with inconsistencies being detected if structure changes can not be automatically applied by environments, semantic constraints are violated, or multiple-developer interference occurs. Environments built using our architecture and associated support tools allow inconsistencies to be presented in a variety of ways.

Augmentation of inconsistency presentations with enacted process model information assists developers to group inconsistencies and understand the reasons for their existence. Some inconsistency presentations can be interacted with to resolve the inconsistency or gain further information about them. Our environments provide mechanisms for developers to configure inconsistency detection, storage, presentation and to some degree interaction, allowing developers to choose the most suitable approach to inconsistency management for different artefacts, tools and process stages. We have demonstrated the usefulness of our diverse inconsistency management approaches through the development and use of various exemplar systems.

We are continuing the development of JComposer systems with improved abstractions for representing constraints, event handling and inter-operation with third-party tools. We are also continuing the development of inconsistency management facilities, such as improved support for tracking, prioritising and monitoring change descriptions which represent structural and semantic inconsistencies. Improvements to the JComposer icon renderings will allow us to present inconsistencies and allow developer interaction with inconsistencies in a wider range of ways. We are also working on utilising our inconsistency management approaches with a wide range of third-party tools, via Java Beans and Active X interfaces. This will make environments utilising our approaches more organisationally feasible for large-scale software development. We are continuing the development of the Serendipity process modelling environment, including capabilities to manage histories of inconsistencies and anticipate trajectories (i.e. future) of inconsistencies. This will provide developers with semi-automated management facilities for inconsistencies for large scale software process enactment which uses knowledge about previous inconsistencies to predict possible future inconsistencies. More sophisticated inconsistency analysis techniques would be useful in grouping and perhaps automatically actioning inconsistencies in many of our environments [56].

Acknowledgments

References

[1]   Altmann, R.A., Hawke, A.N., and Marlin, C.D., "An Integrated Programming Environment Based on Multiple Concurrent Views," *Australian Computer Journal*, vol. 20, no. 2, 65-72, May 1988.

[2]   Amor, R.W. and Hosking, J.G., " Mappings: the glue in an integrated system," in *1st European Conference on product and process modelling in the building industry,* Rotterdam, The Netherlands), A.A. Balkema Publishers, Rotterdam, The Netherlands, 1995, pp. 117-123.

[3]   Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C., "Directions in modelling environments," *Automation in Construction*, no. 4, 173-187, 1995.

[4]   Arefi, F., Hughes, C.E., and Workman, D.A., "Automatically Generating Visual Syntax-Directed Editors," *Communications of the ACM*, vol. 33, no. 3, 349-360, March 1990.

[5]   Arlow, J., Bandinelli, S., Emmerich, W., and Lavazza, L., "Fine grained Process Modelling: An Experiment at British Airways," *Software Process - Improvement and Practice*, vol. 3, no. 2, 105-131, 1997.

[6]   Avrahami, G., Brooks, K.P., and Brown, M.H., " A Two-View Approach to Constructing User Interfaces," *ACM Computer Graphics*, vol. 23, no. 3, 137-146, 1990.

[7]   Backlund, B., Hagsand, O., and Pherson, B., " Generation of Visual Language-oriented Design Environments," *Journal of Visual Languages and Computing* , vol. 1, no. 4, 333-354, 1990.

[8]   Bandinelli, S., Fuggetta, A., and Ghezzi, C., "Process model evolution in the SPADE environment," *IEEE Transactions on Software Engineering*, vol. 19, no. 12, 1128-1144, December 1993.

[9]   Bandinelli, S., DiNitto, E., and Fuggetta, A., "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, 841-865, December 1996.

[10] Belkhatir, N., Estublier, J., and Melo, W.L., *The Adele/Tempo Experience*, Software Process Modelling & Technology. Research Studies Press, 1994, pp. 187-222.

[11] Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D. amd Tong, A.Z., and Valetto, G., "Integrating Groupware and Process Technologies in the Oz Environment," in *9th International Software Process Workshop:The Role of Humans in the Process,* IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.

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

[13] Cayenne Software, *ObjectTeam - Collaborative Object-oriented Development*, http://www.cayennesoft.com/objectteam/, June 1998.

[14] Dannenberg, R.B., " A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors," *Software-Practice and Experience*, vol. 20, no. 2, 109-132, February 1990.

[15] Ebert, J., Suttenbach, R., and Uhe, I., "Meta-CASE in practice: A Case for KOGGE," in *Proceedings of the 9th International Conference on Advanced Information Systems Engineering,* LNCS 1250, Springer-Verlag, Barcelona, Spain, 1997, pp. 203-216.

[16] Emmerich, W., "An Architecture for Viewpoint Environments Based on OMG/CORBA," in *Proceedings of 1996 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96),* ACM Press, San Francisco, 1996, pp. 207-211.

[17] Emmerich, W., "Tool Specification with GTSL," in *Proceedings of the 8th International Workshop on Software Specification and Design,* IEEE CS Press, Schloss Velen, Germany, 1996, pp. 26-35.

[18] Emmerich, W., " CORBA and ODBMSs in Viewpoint Development Environment Architectures.," in *Proceedings of the 4th International Conference on Object-Oriented Information Systems,* Springer Verlag, 1997, pp. 347-360.

[19] Emmerich, W., Arlow, J., Madec, J., and Phoenix., M., "Tool Construction for the British Airways SEE with the O2 ODBMS," *Theory and Practice of Object Systems*, vol. 3, no. 3, 213-231, 1997.

[20] Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process: Continuous Software Process Improvement,* IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.

[21] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., "Inconsistency Handling in Multiperspective Specifications," *IEEE Transactions on Ssoftware Engineering*, vol. 2, no. 8, 569-578, August 1994.

[22] Finkelstein, A., Spanoudakis, G., and Till, D., "Managing Inconsistencies," in *Joint Proceedings of the SIGSOFT'96 Workshops,* ACM Press, San Francisco, October 1996, pp. 172-174.

[23] Fuggetta, A., "A Classification of CASE Technology," *COMPUTER*, vol. 26, no. 12, 25-38, December 1993.

[24] Gamma, E., Helm, R., Johnston, R., and Vlissides, J., *Design Patterns*. Addison-Wesley, 1994.

[25] Graham, T.C.N., "Viewpoints supporting the Development of Interactive Software," in *Proceedings of Viewpoints'96,* ACM Press, San Francisco, 1996, pp. 263-267.

[26] Grundy, J.C., "Multiple textual and graphical views for Interactive Software Development Environments," Ph.D. thesis, University of Auckland, Department of Computer Science, June 1993.

[27] Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., Connecting the pieces, in *Visual Object-Oriented Programming*. Manning/Prentice-Hall, 1995, chap. 11.

[28] 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, Darmsdart, Germany, September 1995, pp. 53-60.

[29] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.

[30] Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with MViews," *International Journal of Applied Software Technology*, vol. 2, no. 3-4, 133-160, 1996.

[31] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Low-level and high-level CSCW in the Serendipity process modelling environment," in *Proceedings of OZCHI'96,* IEEE CS Press, Hamilton, New Zealand, Nov 24-27 1996, pp. 69-77.

[32]   Grundy, J.C. and Venable, J.G., "Towards an integrated environment for Method Engineering," in *Proceedings of the IFIP 8.1/8.2 Working Conference on Method Engineering,* Chapman-Hall, Atlanta, August 26-28 1996.

[33]   Grundy, J.C., Mugridge, W.B., and Hosking, J.G., "A Java-based toolkit for the construction of multi-view editing systems," in *Proceedings of the Second Component Users Conference,* SIGS Publications/CUP, Munich, Germany, July 14-18 1997.

[34]   Grundy, J.C. and Hosking, J.G., "Serendipity: integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, January 1998, 27-60.

[35]   Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Static and Dynamic Visualisation of Software Architectures for Component-based Systems," in *Proceedings of SEKE'98,* IEEE CS Press, San Francisco, June 18-20 1998.

[36]   Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Support for end user specification of workflows, work coordination and tool integration," *Journal of End User Computing*, vol. 10, no. 2, May 1998, 38-48.

[37]   Hart, R.O. and Lupton, G., "DECFUSE: Building a graphical software development environment from Unix tools," *Digital Tech Journal*, vol. 7, no. 2, 5-19, 1995.

[38]   Hill, R.D., Brinck, T., Rohall, S.L., Patterson, J.F., and Wilner, W., "The Rendezvous Architecture and Language for Constructing Multi-User Applications," *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, 81-125, June 1994.

[39]   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. 101-106.

[40]   Interactive Development Environments Inc., *Software thru Pictures 2.4.2*, http://www.ide.com/Products/StP/stp.html, 1997.

[41]   Interactive Software Engineering Inc.,*Eiffel CASE*, http://www.eiffel.com/products/case.html, 1998.

[42]   Kaiser, G.E. and Garlan, D., " Melding Software Systems from Reusable Blocks," *IEEE Software*, vol. 4, no. 4, 17-24, July 1987.

[43]   Kelly, S., Lyytinen, K., and Rossi, M., "Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment," in *Proceedings of CAiSE'96,* Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.

[44]   Kiper, J.D., "A framework for characterisation of the degree of integration of software tools," *Journal of Systems Integration*, vol. 4, 5-32, 1994.

[45]   Krant, R.E. and Streeter, L.A., "Coordination in Software Development," *Communications of the ACM*, vol. 38, no. 3, 69-81, March 1995.

[46]   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*, vol. 1, no. 3, 8-22, 1988.

[47]   Leonhardt, U., Finkelstein, A., Kramer, J., and Nuseibeh, B., "Decentralised Process Modelling in a Multiple-Perspective Development Environment," in *17th International Conference on Software Engineering,* IEEE CS Press, Seattle, Washington, 1995.

[48]   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.

[49]   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.

[50]   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.

[51]   Marlin, C., Peuschel, B., McCarthy, M., and Harvey, J., "MultiView-Merlin: An Experiment in Tool Integration," in *Proceedings of the 6th Conference on Software Engineering Environments,* IEEE CS Press, 1993.

[52]   Medina-Mora, R., Winograd, T., Flores, R., and Flores, F., "The Action Workflow Approach to Workflow Management Technology," in *Proceedings of CSCW'92,* ACM Press, 1992, pp. 281-288.

[53]   Meyers, S., "Difficulties in Integrating Multiview Editing Environments," *IEEE Software*, vol. 8, no. 1, 49-57, January 1991.

[54] Myers, B.A., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *COMPUTER*, vol. 23, no. 11, 71-85, 1990.

[55] Myers, B.A., "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, 347-365, June 1997.

[56] Nuseibeh, B., "Towards a framework for managing inconsistency between multiple views," in *Proceedings of Viewpoints'96,* ACM Press, San Francisco, 1996, pp. 184-186.

[57] Peuschel, B., Schäfer, W., and Wolf, S., "A knowledge-based software development environment supporting cooperative work," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 1, 76-106, 1992.

[58] Poon, W.L. and Finkelstein, A., "Consistency Management for Multiple Perspective Software Development," in *Proceedings of Viewpoints'96,* ACM Press, San Francisco, 1996, pp. 192-196.

[59] Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R., "Dora - a structure oriented environment generator," *IEE Software Engineering Journal*, vol. 7, no. 3, 184-190, 1992.

[60] Rational Corporation, "RationalRose 4.0," http://www.rational.com/, 1997.

[61] Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, 276-285, 1985.

[62] Reiss, S.P., " Working in the GARDEN Environment for Conceptual Programming," *IEEE Software*, vol. 4, no. 11, 16-26, November 1987.

[63] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.

[64] Reiss, S.P., " Interacting with the Field environment," *Software practice and Experience*, vol. 20, no. S1, S1/89-S1/115, June 1990.

[65] Reps, T. and Teitelbaum, T., "Language Processing in Program Editors," *COMPUTER*, vol. 20, no. 11, 29-40, November 1987.

[66] Roseman, M. and Greenberg, S., "Building Real Time Groupware with GroupKit, A Groupware Toolkit," *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 1, 1-37, March 1996.

[67] Sorenson, P.G., Findeisen, P.S., and Tremblay, J.P., "Supporting Viewpoints in MetaView," in *Proceedings of Viewpoints'96,* ACM Press, San Francisco, 1996, pp. 237-241.

[68] Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K., "A Business Process Environment Supporting Collaborative Planning," *Journal of Collaborative Computing*, vol. 1, no. 1, 1994.

[69] TeamWARE Inc., *TeamWARE Flow*, http://www.teamware.us.com/products/flow/, 1996.

[70] Valetto, G. and Kaiser, G.E., "Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments," in *IEEE 7th International Workshop on Computer-Aided Software Engineering,* July 1995, pp. 40-48.

[71] 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,* Lecture Notes in Computer Science, Springer-Verlag, Gold Coast, Australia, 1995, pp. 318-328.

[72] Wasserman, A.I. and Pircher, P.A., "A Graphical, Extensible, Integrated Environment for Software Development," *SIGPLAN Notices*, vol. 22, no. 1, 131-142, January 1987.

[73] Welsh, J., Broom, B., and Kiong, D., " A Design Rationale for a Language-based Editor," *Software - Practice and Experience*, vol. 21, no. 9, 923-948, 1991.

[74] Wilk, M.R., "Change Propagation in Object Dependency Graphs," in *Proceedings of TOOLS US '91,* Prentice-Hall, August 1991, pp. 233-247.

John Grundy holds the BSc(Hons), MSc and PhD degrees, all in Computer Science from the University of Auckland. He is currently Senior Lecturer in Computer Science at the University of Waikato. His research interests include software engineering environments, software process technology, visual languages and program visualisation, component-based software architectures, and computer-supported cooperative work.

John Hosking MemIEEE received BSc and PhD degrees from the University of Auckland. He is currently a Senior Lecturer in Computer Science at the University of Auckland, where he has been employed since 1985. His research interests include software engineering, visual languages, constraint languages, and software development environments.

Warwick B. (Rick) Mugridge is a Senior Lecturer in Computer Science at the University of Auckland, New Zealand. His research interests include visual languages, software engineering, and meta-tools for software development and collaboration. He received a PhD in Computer Science from the University of Auckland in 1990.