International Journal of Applied Software Technology, International Academic Publishing Company, Vol. 2, No. 3/4, pp. 133-160

Constructing Integrated Software Development Environments with MViews

John C. Grundy[1] and John G. Hosking[2]

[1]Dept. of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

[2]Dept. of Computer Science, University of Auckland
Private Bag, Auckland, New Zealand
john@cs.auckland.ac.nz

Abstract: MViews is a new approach to building Integrated Software Development Environments (ISDEs). Graph-based representations of software system data, and multiple views of this data, are kept consistent via graph components responding to descriptions of changes made to other components. This technique supports integrated, bi-directionally consistent graphical (interactively-edited) and textual (free-edited) views of data, ISDE integration, and version control and collaborative software development. An object-oriented framework is specialised to construct new environments. Experience using this framework to build and integrate several ISDEs is discussed.

I. Introduction

Integrated Software Development Environments (ISDEs) provide tools for software management tasks, such as analysis, design, implementation, debugging, maintenance and versioning (Meyers 1991, Dart-et-al. 1987). ISDEs usually support both graphical and textual representations, or views, of parts of a software system (Reiss 1987, Ratcliffe-et-al 1992). Support for many views and view types is often wanted, along with some form of consistency management between views sharing data (Meyers, 1991).

ISDEs require a mechanism for integrating tools. Several kinds of integration are important (Thomas and Nejmeh 1992; Bounab and Godart 1995), including: data integration (use of a common data repository); control integration (propagation of and response to events between tools); presentation integration (consistency of tool user interfaces); and process integration (coordinating the use of tools). Supporting collaboration is a goal of most ISDEs (Kaiser et al. 1987; Magnusson et al. 1993). This involves providing low-level collaborative editing tools, and higher-level work coordination, project management and process modelling tools.

This paper describes MViews, a new approach to building ISDEs which support multiple views of software development, flexible environment integration mechanisms, and both low and high-level collaborative work facilities. We start with a brief discussion of an ISDE illustrating the general requirements this work aims to satisfy. The basic MViews model and its novel approach to the support of multiple consistent views is then described. ISDE integration is illustrated with an example. Collaboration facilities of MViews ISDEs are discussed, and a description of ISDE design and implementation given. Experience with MViews, and comparison with related work is described. A summary concludes the paper.

II. Support for General ISDE Requirements

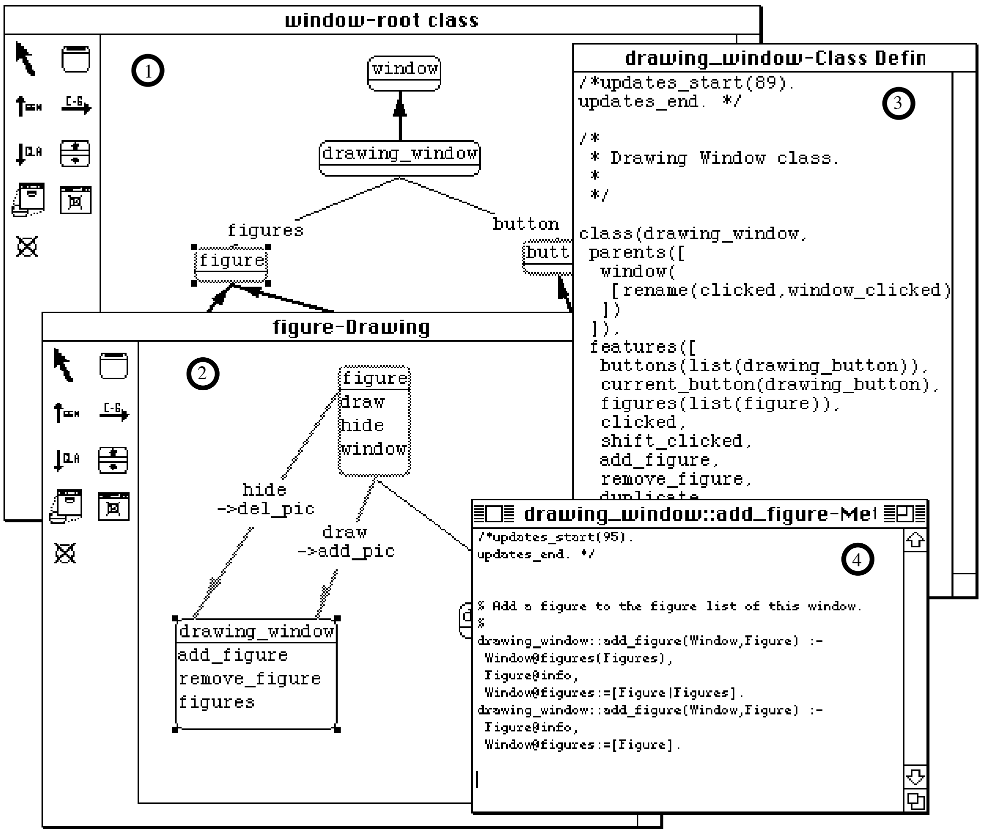
A. An Example Environment

SPE (the Snart Programming Environment) (Grundy et al. 1995a) is an ISDE for Snart (Grundy 1993), an object-oriented extension to Prolog. SPE demonstrates the kind of environment that can be built using MViews. Figure 1 shows a screen dump from SPE during the development of a drawing editor program. The several windows shown reflect different program views. View 1, an analysis-level diagram, shows important inheritance and aggregation (attribute type) structures for drawing program classes; view 2, a design-level diagram, describes method calling protocols for rendering figures in windows. View 3 is a detailed class interface; and view 4 a method implementation. Graphical debugging and textual documentation views are also provided.

Graphical views use an icon and connector representation. They include a palette, to select tools to interactively manipulate the view contents. Textual views are free-edited and parsed. Inter-view navigation is either via menus or an automatically constructed point and click hyper-link system. There is no restriction on the number of views able to be constructed. The contents and layout of each view are under user control.

SPE keeps all data shown in its views consistent under change. New tools need to share this data, need a standard, consistent user interface, and need to respond to events occuring in other tools.

Collaborative software development is supported by C-SPE (Grundy et al. 1995b), an extension providing synchronous, semi-synchronous and asynchronous view editing, plus version control and merging facilities. A work coordination and process modelling tool (Grundy et al. 1995c) allows developers to define appropriate software processes and work plans, and to coordinate their work using shared plan and process views.



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

/*
 * Drawing Window class.
 *
 */

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

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


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

Figure 1 A screen dump from the Snart Programming Environment (SPE).

B. General ISDE Requirements

Integrated Software Development Environments, such as SPE, require several key capabilities, including:

- flexible software system data representation, supporting for both fine-grained and coarse-grained software components (Ratcliffe et al. 1992);
- semamantic constraints between software components (Reps and Teitelbaum 1987);
- support for a diverse range of editable, graphical and textual views of software system data (Reiss 1987; Ratcliffe et al. 1992);
- full consistency (or inconsistency) management between all software components and views (Meyers 1991; Ratcliffe et al. 1992; Grundy et al. 1996a);
- tool integration, including data (repository), control (event), presentation (user interface), and process (methods and procedure) integration (Thomas and Nejmeh 1992; Bounab and Godart 1995);
- and cooperative work support, including collaborative editors, version control, merging and configuration management, and software process modelling facilities (Grundy et al. 1995c).

We have developed MViews to provide a framework supporting such capabilities.

C. An Overview of MViews

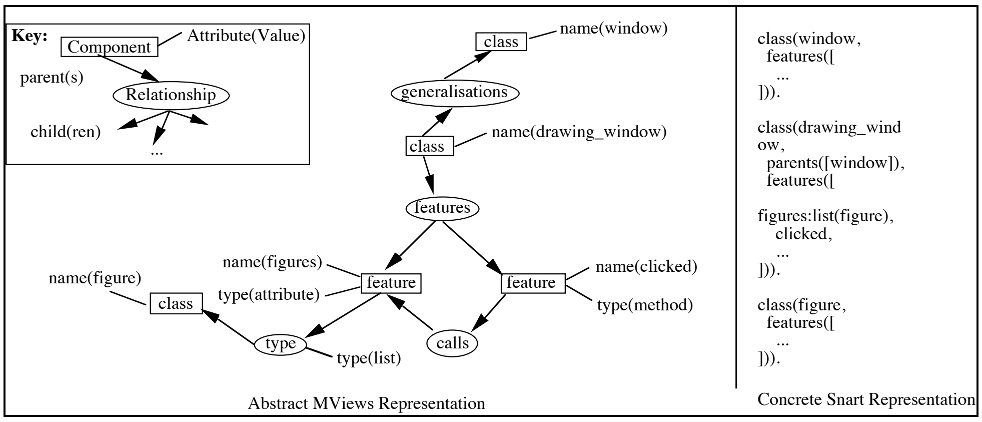
We briefly overview MViews' support for these general requirements of ISDEs. The core of ISDEs like SPE is the representation of the shared data. MViews represents this data using a graph-based form, with syntax, semantic attribute values and multiple views represented as graph components. View graphs are rendered in graphical or textual forms, and modified by user operations. View graph changes are translated into operations on the underlying graphs, with other views updated to maintain consistency. Consistency management is via a novel change propagation mechanism. Whenever a graph component is modified, a description of the change to the component is propagated to all related graph components. These respond by updating their own state to maintain consistency. The response mechanism is often abstracted into the relationships between graph components, resulting in a flexible, reusable consistency management mechanism.

Storage of change descriptions by graph components supports a wide range of other ISDE functions, including a generic undo/redo facility, modification histories, version control and merging, tool integration, and collaborative software development. Flexible environment integration and extensibility are also supported. Broadcasting change descriptions between multiple developers' environments supports both low-level collaborative editing and higher-level work coordination and process modelling and enactment.

MViews environments are built by specialising an object-oriented framework, which provides a consistent, integrated user interface for all tools. Tools not built using MViews can be integrated by building data and event translator views.

III. MViews Data and Multiple Views

A. Repository Data Representation

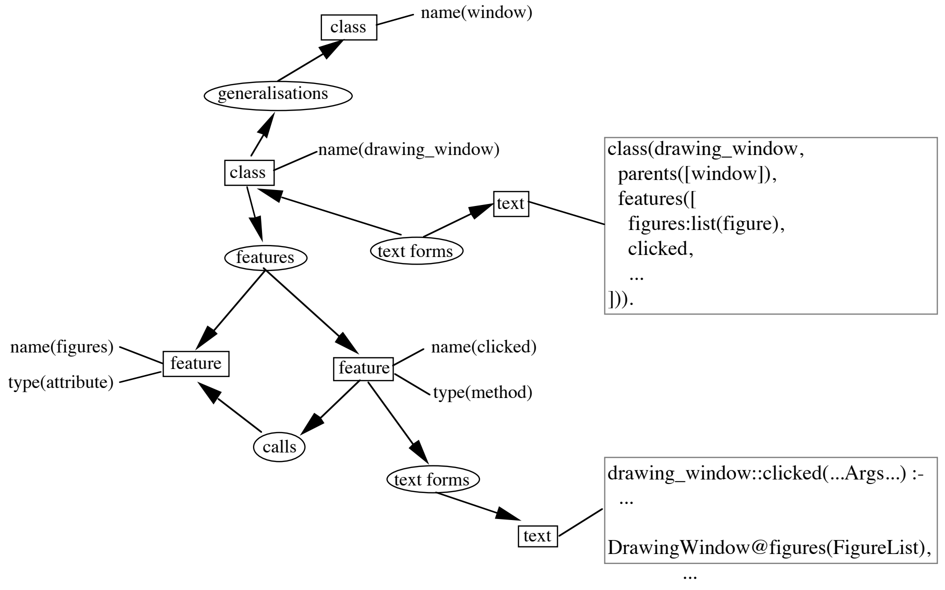


```
class(window,
  features([
   ...
])).

class(drawing_wind
ow,
  parents([window]),
  features([

figures:list(figure),
   clicked,
   ...
])).

class(figure,
  features([
   ...
])).
```

Figure 2 Representing SPE data using MViews program graphs.

MViews represents ISDE data as a collection of (possibly disjoint) directed graphs, thus supporting both tree-based textual languages and graph-based visual languages. Software system data is represented as *components* (nodes) connected by *relationships* (labelled edges). Each component has associated *attributes* (name/value pairs). Figure 2 shows an MViews graph for part of the SPE drawing program of Figure 1. Relationships are specialised forms of components that can be connected to other relationships and have attributes. We collectively refer to all graph nodes and edges as components.

MViews graphs represent both system structure and semantic values. For example, the full interface (signature) of a class in SPE is calculated from attributes and methods the class defines and those it inherits. This interface is represented as a relationship from the class to components representing each calculated interface feature.



```
class(drawing_window,
  parents([window]),
  features([
    figures:list(figure),
    clicked,
    ...
])).
```

```
drawing_window::clicked(...Args...) :-
  ...

DrawingWindow@figures(FigureList),
        ...
```

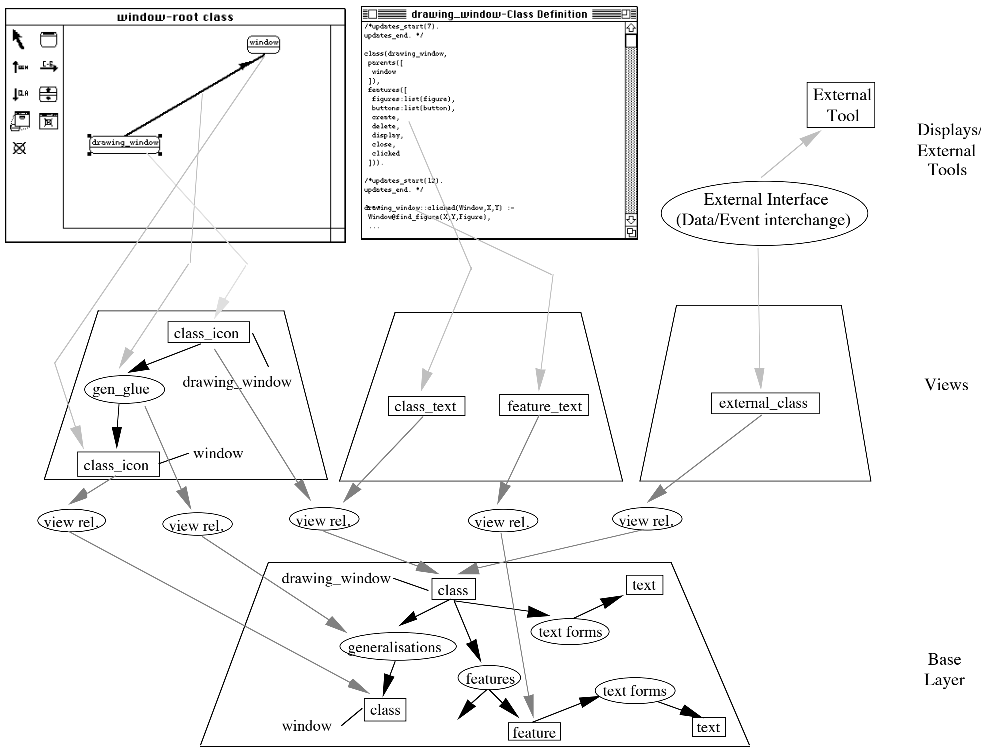
Figure 3 Coarse and fine-grained representations.

Fine-grained software system representation uses the graph structure, and coarse-grained uses *text components*. The latter are nodes containing a sequence of characters. For example, in Figure 3, the `drawing_window` class and `clicked` method have text components corresponding to the formatted text of their concrete representations. Data needing individual editing or semantic values is represented by graph components. Free-edited data used only as a single component is represented in a coarse-grained fashion, for greater efficiency. For example, method code and class documentation are represented only as text components in SPE. Class interfaces are represented as both text components and fine-grained graph components. This allows the format of the class definition, such as comments and user-defined layout, to be kept. Consistency is maintained between the two representations by the mechanism described below.

B. Multiple Views

MViews uses a three-layer architecture to provide multiple views (Figure 4). A *base layer* (the ISDE repository) provides a shared graph representation of the software system. *Views* are graphs representing information needed for each view, i.e. base layer components have matching view components. *Displays* act as both renderers and interactive editors for view components. External views provide an interface mechanism for tools not implemented using MViews.

View relationships connect view and base components, permitting view components to access base component data and supporting bi-directional propagation of changes between base and view. MViews utilises free form editing of textual views, allowing existing editors to be used for textual programming, and because this interaction style is preferred by programmers (Welsh et al. 1991). MViews uses interactive editing for graphical views; view components are modified by direct manipulation of their renderings. This, again, is preferred by users (Arefi et al. 1990). This contrasts with other ISDEs which use restrictive structure-oriented editing approaches for all views (although MViews can support such views).



```
/*updates_start(7).
updates_end. */

class(drawing_window,
 parents([
  window
 ]),
 features([
  figures:list(figure),
  buttons:list(button),
  create,
  delete,
  display,
  close,
  clicked
 ])).

/*updates_start(12).
updates_end. */

drawing_window::clicked(Window,X,Y) :-
 Window@find_figure(X,Y,Figure),
  ...
```

Figure 4 Multiple views of an SPE program in MViews.

C. Graph Operations and Change descriptions

The MViews graph structure is based on the Change Propagation and Response Graph (CPRG) model. Descriptions of changes made to CPRG components are propagated to related components to maintain inter-component consistency. We briefly describe use of this mechanism in MViews environments; further details are in (Grundy et al. 1996a).

Graphs are modified by *operations* applied to components. Components support basic operations, add-component, establish-relationship, update-attribute, etc. Operations are of the form `Comp.Op(Arg1,Arg2,...)` where component `Comp` has operation `Op` applied with parameters `Arg1, Arg2,` etc. More complex *macro operations* can be defined, made up of sequences of basic graph operations.

When a component is modified, components dependent on its state must be updated to maintain consistency. MViews components use the CPRG *change description* mechanism to broadcast a description of an operation's effect on a component to all its attached relationships. Change descriptions are tuples of the form `UpdateKind(Component,Value1, Value2,...)`, where: `UpdateKind` is the operation applied (update attribute, establish relationship, delete component, etc.); `Component` is the modified component which generated the change description; and `Value1` etc are operation-specific values describing exactly the change made to the component. For example a `Comp.update(Name, NewValue)` operation (i.e. update of attribute `Name` of component `Comp`) generates the change description `update(Comp,Name,OldValue,NewValue)`, where `OldValue` is the previous value of `Name` for `Comp`.

Change descriptions are first propagated to the relationships attached to the modified component, (its *dependents*). These relationships respond by: updating their own states, or those of other connected components, by applying operations; propagating the change description to other components participating in the relationship; or ignoring the change. Components receiving change descriptions can interpret them in any way and may apply further operations to maintain consistency with the changed component.

D. Consistency Management

An example of consistency management between MViews components and views (implementing SPE) is shown in Figure 5. 1) An SPE user edits a graphical view component, applying operations to it, or a textual view parser generates operations by comparing updated view text to base information. 2) Resulting change descriptions are propagated to the view component's dependents, including its view relationship. 3) The view relationship translates the view component change into operations on its base component. If the update is view-specific, for example changing an icon's location or text font, the base component is unaffected by the change, so the view relationship does not propagate it. 4) Base component operations generate change description(s) which 5) are propagated to the base component's dependents, including all of its view relationships. 6) The view relationships translate the base component change descriptions into operations on their view components. If the view component is unaffected by the base component change, for example the view component doesn't use an updated attribute, the view relationship does not propagate the change. 7) Updated view components re-render their displays; external view components send messages to external tools or translate changed data into external tool data.

MViews' generic *view relationship* translates changes to base component attributes to changes to view component attributes of the same name (and vice-versa). This behaviour can be modified by defining specialised view relationships supporting, for example, automatic addition of view components when base components are added. Generic *aggregation relationships* propagate change descriptions on child components to the relationship parent. For example, changes to SPE feature components are propagated to the owning class component. Generic *attribute dependency relationships* between components use change descriptions for efficient incremental attribute recalculation. For example, addition of a new feature to a class causes recalculation of a class interface. MViews, via CPRGs, thus provides the speed and flexibility of an operational approach to the inter-component consistency problem, yet al. lows generic, relationship-type-specific constraints to be easily specified and reused.

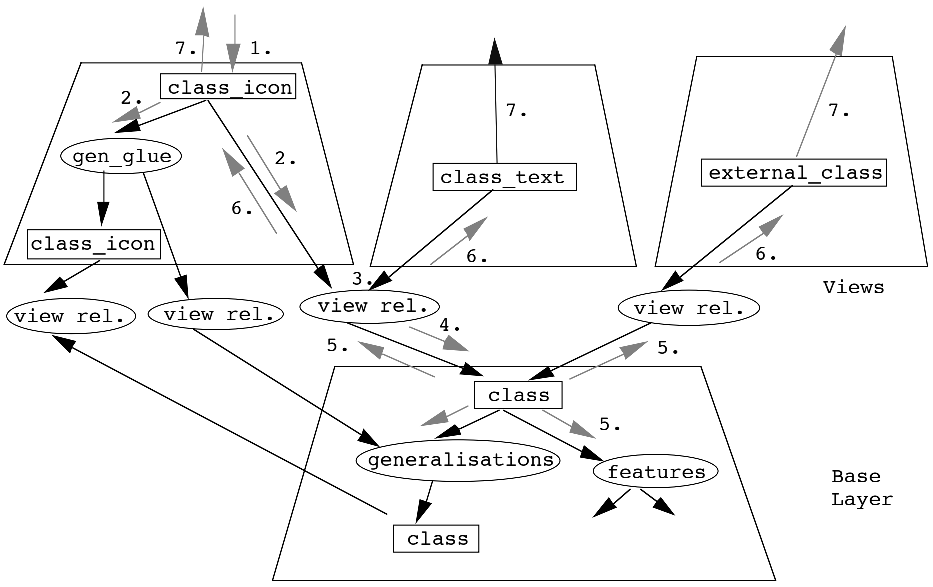


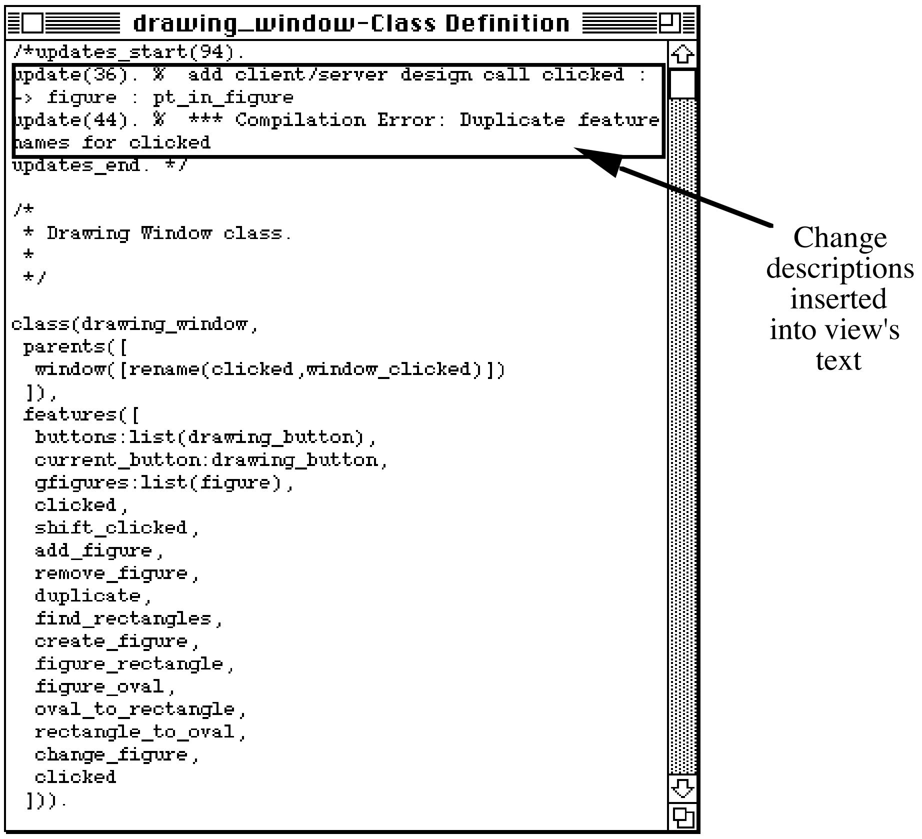
Figure 5 Change propagation between components and views.

E. Multiple View Consistency

As any base layer component may be represented in several views, an ISDE like SPE must keep these different views consistent. To achieve this:

- changes are propagated to all affected views, so no inconsistent information is used
- views are updated automatically (if possible)
- changes made to a view that can't be automatically applied to other views (eg propagation of a design view change to a code view) are indicated in some way

For example, after editing an SPE graphical view, other graphical views are usually automatically updated. If a method is renamed in one view, other graphical representations of the method are updated automatically. Changes such as renaming classes or features, adding or deleting features, and changing feature types can be automatically applied to textual views. Other changes cannot. For example when a client-supplier relationship (to be implemented by a method call) is added between classes in a graphical design view, the required change to a textual view cannot be automatically inferred. In this case, a change description is inserted into the view's text, indicating it is affected, as seen in figure 6. These unparsed change descriptions act as a visual cue for the programmer to manually implement the change (for example, by adding an appropriate method call to the view).

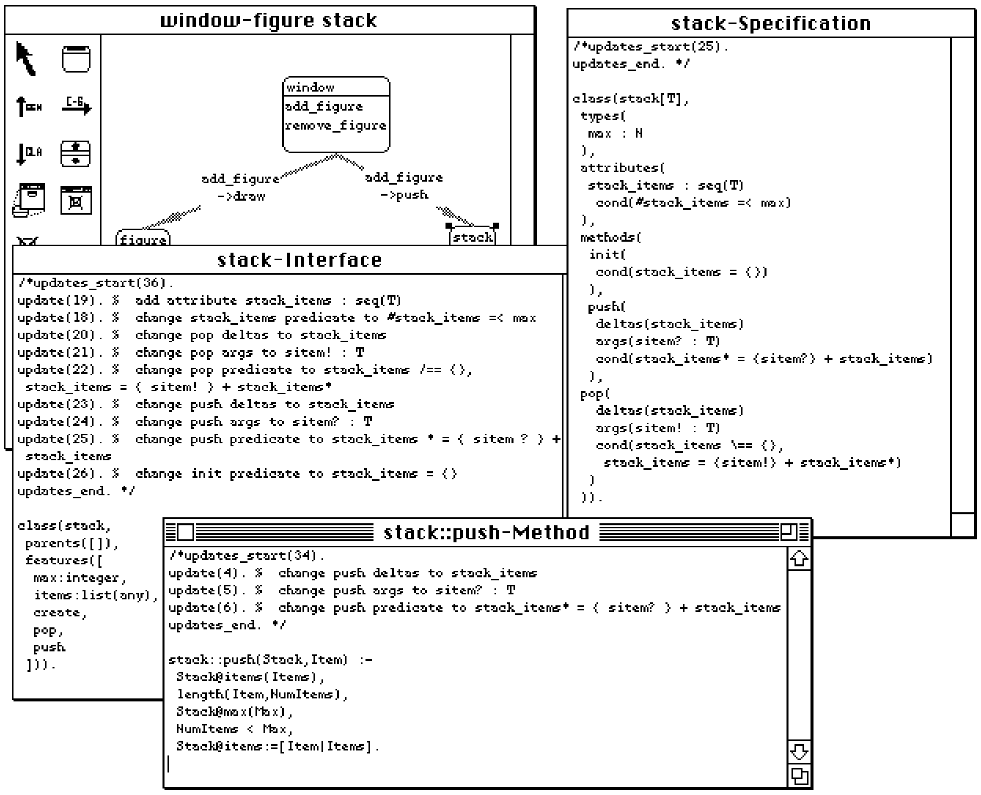

drawing_window-Class Definition
Change descriptions inserted into view's text

```
/*updates_start(94).
update(36). %  add client/server design call clicked :
-> figure : pt_in_figure
update(44). %  *** Compilation Error: Duplicate feature
names for clicked
updates_end. */

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
 parents([
  window([rename(clicked,window_clicked)])
 ]),
 features([
  buttons:list(drawing_button),
  current_button:drawing_button,
  gfigures:list(figure),
  clicked,
  shift_clicked,
  add_figure,
  remove_figure,
  duplicate,
  find_rectangles,
  create_figure,
  figure_rectangle,
  figure_oval,
  oval_to_rectangle,
  rectangle_to_oval,
  change_figure,
  clicked
 ])).
```

Figure 6 Textual view consistency in SPE.

Changes from graphical view to graphical view, and from textual view to textual view, can also be presented in this way. For example, figure 7 shows an Object-Z-like view in SPE being kept consistent with a method implementation view (Grundy and Hosking 1995a). Both views are textual, and changes to one type of view are shown in the other type by change descriptions.

Environments can also present inconsistencies in more context-dependent ways. Figure 8 is a screen dump from MViewsDP, an interface builder supporting graphical and textual dialog specification (Grundy et al. 96a). The dialog control button ‘Ok’ in the graphical view has been shifted, reflected in the textual view header by a change description. This change description can also be shown in a dialog associated with the graphical view. A semantic inconsistency resulting from the Ok button’s border overlapping that of its enclosing dialog box has been created, however, which must be resolved. This inconsistency is indicated by shading the Ok button icon, so the user of the environment is drawn to this component. Other presentation techniques can be utilised to highlight inconsistencies, including shading and colouring graphical icons, changing font, style, size or colour of text, or more dynamic techniques, such as blinking affected icons (Grundy and Hosking 1996b).



```
/*updates_start(36).
update(19). %  add attribute stack_items : seq(T)
update(18). %  change stack_items predicate to #stack_items =< max
update(20). %  change pop deltas to stack_items
update(21). %  change pop args to sitem! : T
update(22). %  change pop predicate to stack_items /== {},
 stack_items = { sitem! } + stack_items*
update(23). %  change push deltas to stack_items
update(24). %  change push args to sitem? : T
update(25). %  change push predicate to stack_items * = { sitem ? } +
 stack_items
update(26). %  change init predicate to stack_items = {}
updates_end. */

class(stack,
 parents([]),
 features([
  max:integer,
  items:list(any),
  create,
  pop,
  push
 ])).
```

```
/*updates_start(25).
updates_end. */

class(stack[T],
 types(
  max : N
 ),
 attributes(
  stack_items : seq(T)
   cond(#stack_items =< max)
 ),
 methods(
  init(
   cond(stack_items = {})
  ),
  push(
   deltas(stack_items)
   args(sitem? : T)
   cond(stack_items* = {sitem?} + stack_items)
  ),
 pop(
   deltas(stack_items)
   args(sitem! : T)
   cond(stack_items \== {},
    stack_items = {sitem!} + stack_items*)
  )
 )).
```

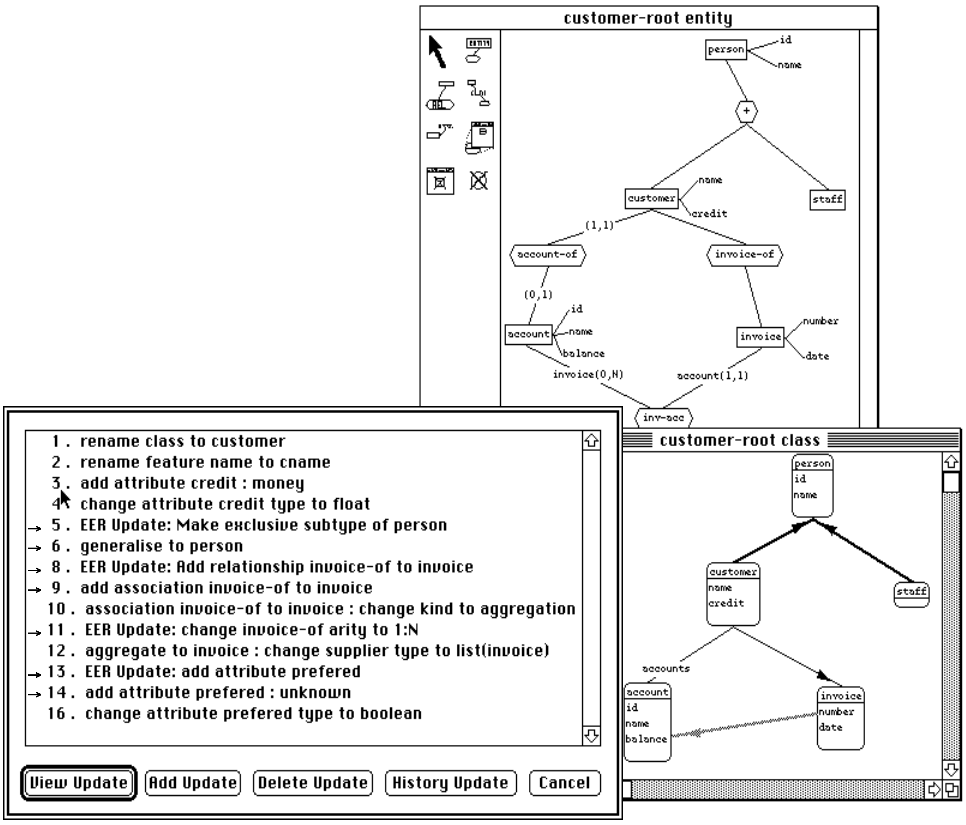
```
/*updates_start(34).
update(4). %  change push deltas to stack_items
update(5). %  change push args to sitem? : T
update(6). %  change push predicate to stack_items* = { sitem? } + stack_items
updates_end. */

stack::push(Stack,Item) :-
 Stack@items(Items),
 length(Item,NumItems),
 Stack@max(Max),
 NumItems < Max,
 Stack@items:=[Item|Items].
```

Figure 7 Keeping textual views consistent with one another.

IV. Tool Integration

In order to maximise reuse of ISDE tools, techniques for integrating different tools into an ISDE have become an important research area. The MViews graph structure, base and view layers, and its change description propagation mechanism supports flexible data and control ISDE integration.

Figure 8 Indicating inconsistencies in graphical views.



```
   1 .  rename class to customer
   2 .  rename feature name to cname
   3 .  add attribute credit : money
   4    change attribute credit type to float
→  5 .  EER Update: Make exclusive subtype of person
→  6 .  generalise to person
→  8 .  EER Update: Add relationship invoice-of to invoice
→  9 .  add association invoice-of to invoice
  10 .  association invoice-of to invoice : change kind to aggregation
→ 11 .  EER Update: change invoice-of arity to 1:N
  12 .  aggregate to invoice : change supplier type to list(invoice)
→ 13 .  EER Update: add attribute prefered
→ 14 .  add attribute prefered : unknown
  16 .  change attribute prefered type to boolean
```

Figure 9 Integrated OOA/D and EER views with bi-directional consistency management.

We illustrate ISDE integration with MViews, using OOEER, which integrates SPE and MViewsER, an Entity-Relationship diagraming and textual relational schema modelling tool. OOEER is thus an integrated environment for OOA/D and EER modelling (Grundy and Venable 1995). Figure 9 shows a screen dump from OOEER. The OOA/D views are kept consistent with all changes to the EER views, and vice-versa, even when a direct translation is not possible by the environment. The dialog shown holds change descriptions (the "modification history") for the customer OOA class. The change descriptions highlighted by '→' were actually made to the EER view (diagram) and automatically translated into OOA/D view updates (where possible) by OOEER. Unhighlighted items were made by the designer to the OOA view to fully implement "indirect" translations that could only partially by implemented by OOEER.

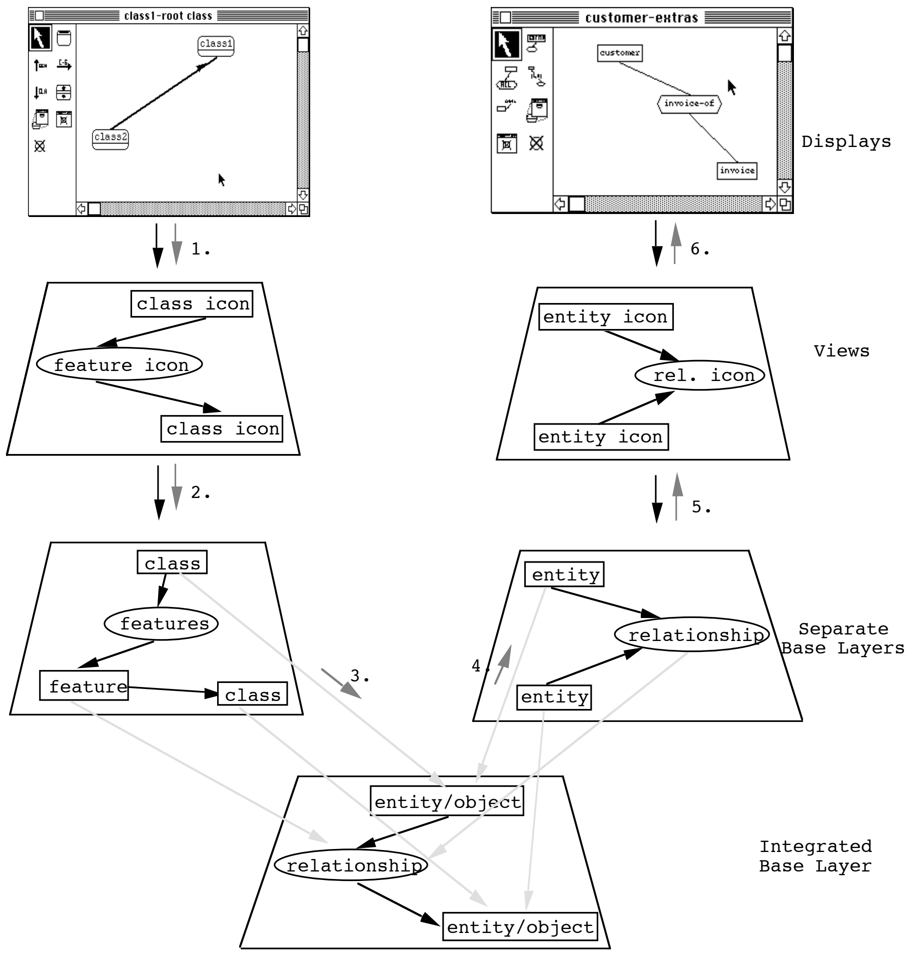


Figure 10 Integrating SPE and MViewsER to produce OOEER.

OOEER integration was achieved via an additional repository graph level below the base layers of SPE and MViewsER (Grundy and Venable 1995). This layer translates changes (where possible) between the different notations, and notifies tools where automatic translations are not possible. Neither SPE nor MViewsER required any significant change to achieve this integration. Figure 10 shows an example of the structure of OOEER. When an SPE view is edited (1), the modification is translated into SPE repository updates (2), generating change descriptions. *Inter-repository relationships* are sent change descriptions, and respond by updating the integrated repository (3).

When integrated repository components change, the inter-repository relationships to MViewsER's repository components translate these change descriptions into updates on MViewsER repository components (4). Indirect translations are defaulted where possible and change descriptions displayed in views. Both SPE and MViewsER keep their multiple views consistent (5 and 6).

Inter-repository relationships are implemented as specialisations of MViews' generic many-to-many relationships. On receiving a change description, inter-repository relationships determine the change to make to related components. This might be a simple update (automatic mapping), partial update (semi-automatic mapping) or storage of the change description against the affected component(s) (no automatic mapping possible). Using CPRG change description composition facilities (Grundy et al. 96a), relationships can wait for receipt of several change descriptions, allowing more complex translations mapping several updates from another model. CPRGs provide lazy processing capabilities (Grundy et al. 96a) which are used to minimise response time delay. Much of the translation and consistency management is performed on-demand when a view is selected for editing, by caching change descriptions in an integrated data dictionary.

Interrepository relationships can be added directly between components in different tool repositories, but this approach has the disadvantage of requiring further relationships if other tools are to be integrated at a later date (Grundy and Venable 1995).

This example illustrates how MViews environments support both data integration (via hierarchical, integrated repositories) and control integration (via the change description broadcasting between tools). User interface integration is achieved by building tools from a standard library of object-oriented classes (see Section 6).

V. Collaborative Software Development

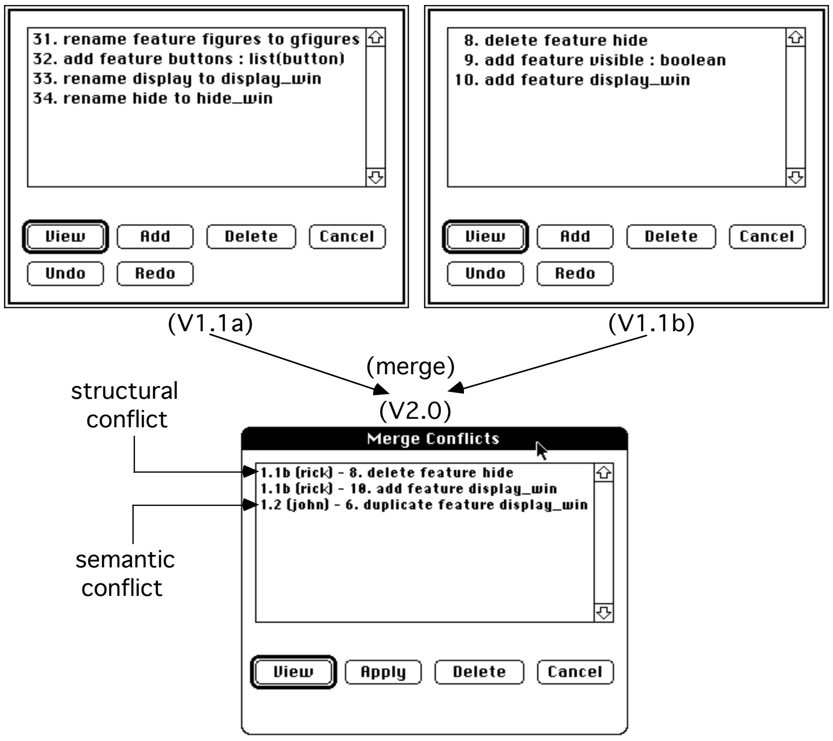
Computer-Supported Cooperative Work (CSCW) systems may use a multi-view editing approach to sharing and modifying information (Ratcliffe et al. 1992; Reiss 1990a; Meyers 1991). Inconsistencies between views then become more difficult to resolve as different users are affected. MViews provides support for multiuser asynchronous, semi-synchronous and synchronous view editing (Grundy et al. 1995a). We have recently developed a high-level work coordination and process modelling tool which supports the coordinated use of integrated MViews tools (Grundy et al. 1995c; Grundy and Hosking 1996a).

A. Versioning and Asynchronous Development

Change descriptions can be used to describe changes between different versions of views i.e. they form a view modification history (Grundy et al. 96a). A non-sequential undo/redo facility is provided by reversing or reapplying view changes, and this supports version merging (Grundy et al. 1995a; Grundy et al. 96a). An example merge from SPE is shown in figure 11. The top two dialogues contain change descriptions for two versions of a software component. The bottom dialogue shows structural and semantic conflicts in the merged version, presented to the developer as a sequence of merge conflict change descriptions.

B. Synchronous and Semi-synchronous Editing

Semi-synchronous collaboration is supported by broadcasting change descriptions between different users' environments. These are presented in a view or dialog. Figure 12 is an example of semi-synchronous view editing in SPE. Synchronous collaboration allows users to view and edit the same version of a view, with fine-grained locking on view components (Grundy et al. 1995a).



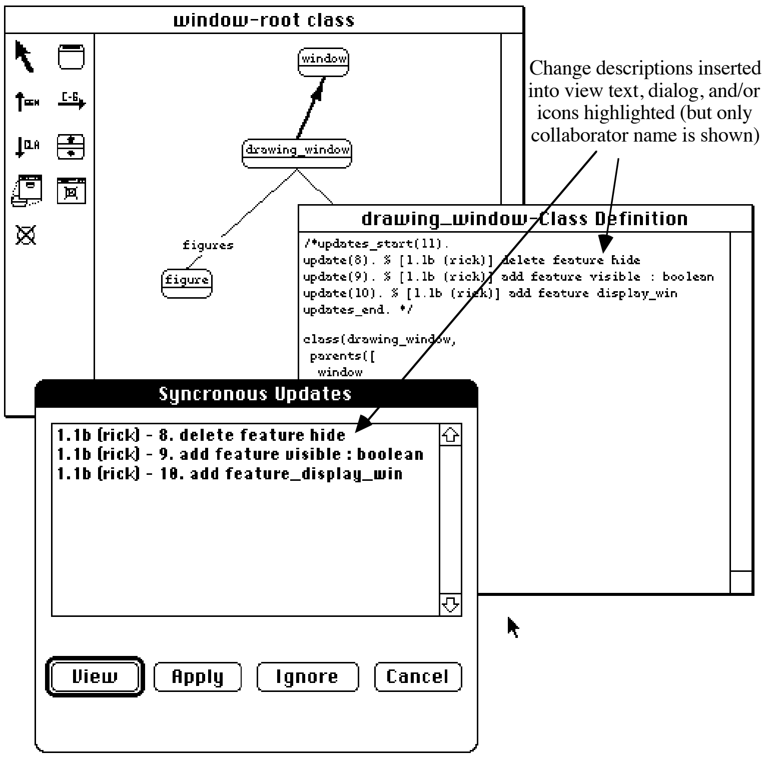
```
31. rename feature figures to gfigures
32. add feature buttons : list(button)
33. rename display to display_win
34. rename hide to hide_win
```

```
 8. delete feature hide
 9. add feature visible : boolean
10. add feature display_win
```

Merge Conflicts

```
1.1b (rick) - 8. delete feature hide
1.1b (rick) - 10. add feature display_win
1.2 (john) - 6. duplicate feature display_win
```

Figure 11 Version merging with MViews.


window-root class
window
drawing_window
figures
figure
Change descriptions inserted into view text, dialog, and/or icons highlighted (but only collaborator name is shown)
drawing_window-Class Definition
Syncronous Updates
View
Apply
Ignore
Cancel

```
/*updates_start(11).
update(8). % [1.1b (rick)] delete feature hide
update(9). % [1.1b (rick)] add feature visible : boolean
update(10). % [1.1b (rick)] add feature display_win
updates_end. */

class(drawing_window,
 parents([
  window
```

```
1.1b (rick) - 8. delete feature hide
1.1b (rick) - 9. add feature visible : boolean
1.1b (rick) - 10. add feature_display_win
```

Figure 12 Semisynchronous view editing via change description broadcasting.

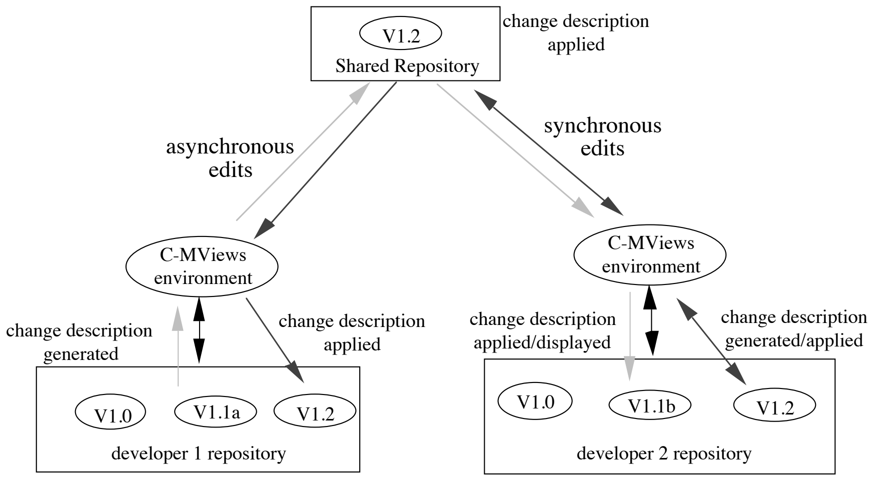


Figure 13 Supporting collaborative editing in MViews ISDEs.

Figure 13 shows the architecture used to support collaborative view editing in MViews environments. A central repository provides shared access to multiple versions of views and components, supporting asynchronous software development. For semi-synchronous editing, broadcast change descriptions are received by other developers environments and cached in special version records. These are presented in dialogs or textual view headers to inform other collaborators of changes. The change descriptions can be incrementally merged into the receiving developers' views using the merging techniques of 5.1. For synchronous editing, no developer "owns" the shared version of a view. All updates attempted are sent to the central server which updates the shared version and broadcasts change description(s) to all collaborators, whose views are then re-rendered. The server uses fine-grained locking so only one edit of the same component is accepted at a time.

C. Work Coordination and Software Process Modelling

ISDEs are large cooperative work systems, and so require support for process modelling, enactment, and coordination of cooperative work (Krant and Streeter 1995). The Serendipity environment provides this support for MViews ISDEs (Grundy and Hosking 1996a; Grundy et al. 1995c). Serendipity provides visual languages to describe process models and specify flexible event processing for these models. Process model animation allows collaborators to be aware of the work contexts of colleagues. Information about the current enacted process stage is attached to change descriptions, recording the context of work. These changes are also stored by the current enacted process stage, allowing collaborators to review the work history for process stages. Serendipity's visual event processing language allows users to specify rules and actions triggered by enactment events, process or work artefact modifications, or tool events.

Figure 14 shows a simple Serendipity process model for updating a software system ("m1:model1-process"). The basic notation here is derived from Swenson's Visual Planning Language (VPL), although VPL does not support role, artefact or tool modelling, nor arbitrary event filtering and actioning (Swenson 1993). Several *stages* describe steps in the process of modifying a software system, with each stage containing a *unique id*, the *role* which will carry out the stage, and the *name* of the stage. Enactment *event flows* link stages. If labelled, the label is the *finishing state* of the stage the flow is from (e.g. "finished design"). The shading of the "m1.2:implement changes" stage indicates that multiple implementers can work on this stage (i.e. the stage has multiple subprocess enactments). Other items include *start stages*,*finish stages*, *AND* stages, and *OR* stages (empty round circle). Underlined stage IDs/roles mark presence of a *subprocess* model. For example, "m1.1:design changes" has a subprocess model ("m1.1:design changes-subprocess"). The italicised "check out design" stages in this subprocess model indicate stages reused from a *template* process model.

During a project, stages in the model are *enacted*, with such stages highlighted by colour and shading (Grundy and Hosking 1996a). As a stage completes in a given finishing state, event flows with this state name (or no name) activate to enact linked stages. Enactments of stages are recorded, as are work artefact changes made while a stage is a user's *current enacted stage* (i.e. the user's work context). Change descriptions are augmented with work context information,. Stored change descriptions and those presented to collaborators thus document the work context they were carried out in.

Serendipity has been integrated with SPE and MViewsER, without modification to these pre-existing environments. MViews ISDEs and Serendipity are integrated by routing change descriptions to the current enacted Serendipity stage. This stores the change description in its artefact modification history and also forwards it to interested collaborating users. Interest in change descriptions and enactment events is determined by a filter and action visual language (Grundy and Hosking 1996a). Interest may be hierarchical, so interest in a stage includes interest in its subprocess. Actions may forward change descriptions to other users and/or present them to these users in views, dialogues, or via shading view items. Actions may also enact, finish or modify other stage components, or may carry out other arbitrary processing.

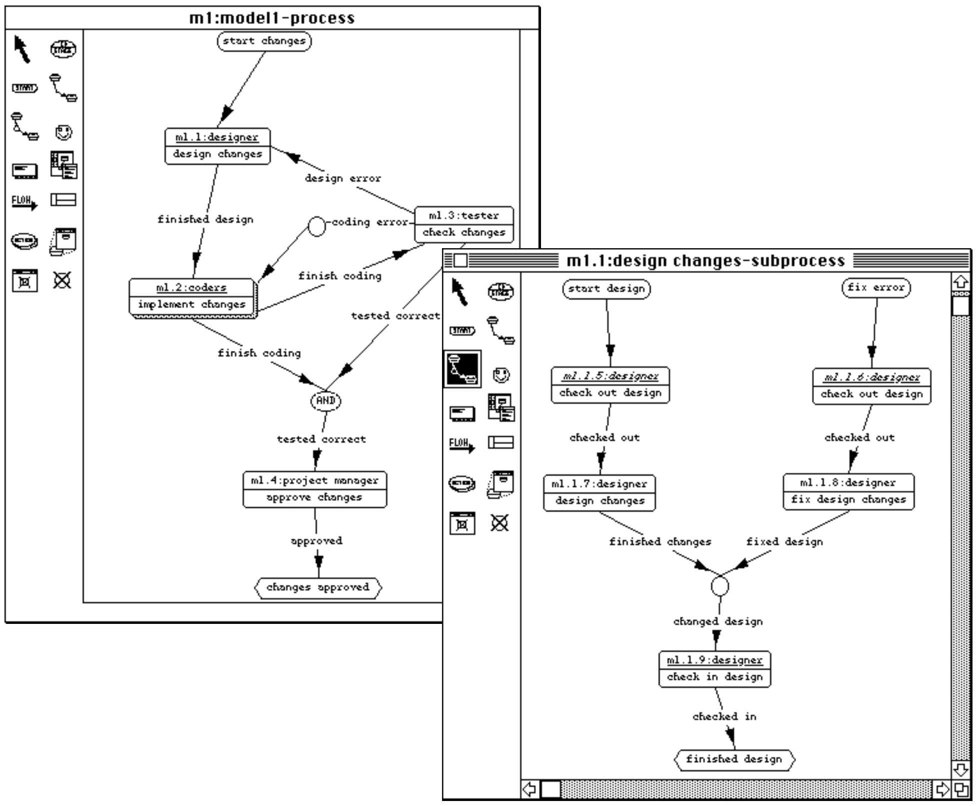


Figure 14 Simple software process models in Serendipity.

VI. Constructing ISDEs with MViews

MViews environments are constructed by reusing and refining classes from an object-oriented framework. The MViews class hierarchy defines general component data and behaviour, relationships, base and view/display components, and various supporting data structures. Classes for a new ISDE, such as SPE, are specialised from classes in this hierarchy, with most component data and behaviour being inherited.

ISDE implementers must define application-specific component data and functionality. For example, SPE classes specify class names, kind (abstract or concrete), and various inter-component relationships (the class's generalisations, features, associations, etc.). Display components require specification of rendering (e.g. how a class icon or an association glue is to be drawn), or appropriate parsing/unparsing grammars for textual views. Default editing facilities for views and view components are built in, but can be specialised by ISDE implementers. All view edits are translated into operations on view components by calling operation methods.

The CPRG change description propagation mechanism is built into the MViews framework, and the ISDE implementer need only define response methods. Many of these are defined in generic relationhips (Grundy et al. 96a) which ISDE developers simply reuse. Application classes can, however, override any response method to specify additional or more specialised functionality.

MViews framework classes automatically handle most graphical view updates, as the structure of rendered graphical view components is held in the view layer. It is more difficult to keep free-edited textual views consistent with changes to other views, as the structure of the view's components must be recovered via parsing. By default, all change descriptions potentially affecting the text of a view are automatically added to the view's text in the header section. ISDE implementers can, however, define textual view component methods which update their view's text. This uses a process we call *incremental unparsing*, whereby regular expressions are specified to incrementally parse and extract token data from the textual view. These tokens are modified as necessary to update the view.

MViews environments use CPRG change descriptions to implement many other ISDE facilities. As change descriptions are automatically generated by components after modification, they are used to record changes to view components for an undo/redo facility and for modification histories and version control. Their propagation can be delayed to support lazy consistency management, and discrete change descriptions can be composed into higher-level change descriptions.

MViews has been implemented using Snart, itself implemented in LPA MacProlog. Snart classes implement component and relationship types, class attributes implement component attributes and methods implement operations. Simple link relationships are implemented by object list attributes for efficiency. Change descriptions are represented by Prolog terms, and methods used to process change description terms are written in a declarative style, by specifying change description patterns to respond to and methods for response.

Graphical component renderings are defined using LPA's Graphics Description Language (GDL). MViews provides an object-oriented interface to GDL, and building-blocks for constructing graphical editors. The regular expressions used in incremental unparsing are interpreted by Snart methods. Definite Clause Grammars (DCGs) are used to provide parsing facilities to translate textual view changes into operations on base components. MViews uses a standard Macintosh text window editor for displaying and editing textual views.

Component persistency is supported via persistent Snart objects. Snart objects may be dynamically saved to and loaded from a high-performance, single-user persistent object store, requiring no programming by ISDE implementers. Text component data is saved as complex object attributes, and an in-core caching scheme means the performance of persistent Snart objects is almost the same as dynamic objects (Grundy 1993).

A C++ port of MViews is being implemented. This runs much faster than the Snart framework but its functionality is as yet less developed.

VII. Experience with MViews

In addition to SPE, MViewsDP, MViewsER and OOEER, we have developed several other ISDEs using MViews. MViewsNIAM provides NIAM modelling views, and has been integrated with MViewsER to produce NIAMER (Venable and Grundy 1995). ViTABaL (Grundy and Hosking 1995b) is a visual tool abstraction language. We have also integrated MViewsDP, OOEER, and NIAMER to produce an integrated information systems engineering environment (Grundy et al. 1996).

MViews has also been used by other researchers. Cerno-II (Fenwick et al. 94) is a graphical debugger complementing SPE for visualising a running Snart program, providing graphical object representations, and visualisation of collection structures, method calls and timing diagrams. EPE is an environment for constructing EXPRESS specifications and corresponding EXPRESS-G diagrams (Amor et al. 1995). Hyper-Pascal (Lyons et al. 1993) is a visual Pascal-like language which provides a variety of graphical programming views together with textual views for input/output format specification. Skin provides a visual/textual functional language for constructing flexible user interface components and prototyping their execution (Hosking et al. 1995).

SPE and MViews have been used to develop substantial, useful, and practical ISDEs and software systems. The largest ISDEs, SPE and EPE, provide reasonable performance for developing quite large applications. For example, the largest systems so far modelled in SPE have been MViews and SPE themselves, which together consist of over 60 Snart classes with 1100 class attributes and methods. 25 graphical views and nearly 200 textual views make up the complete system definition in SPE. EPE has been used to model similar sized systems (60-70 classes in 20-30 views representing a generic model of buildings). EPE was constructed by specialising both SPE and Cerno-II, which involved the addition of several relationships between classes not modelled in SPE and changes to the rendering of both graphical and textual forms of classes. The developer of EPE did not need to modify the SPE framework, but only needed to specialise SPE classes to redefine renderings and add extra attributes and relationships.

Feedback from users of SPE and other MViews ISDEs indicates they find the degree of integration in MViews environments useful. They like the way views are kept consistent, particularly the way potential effects on textual view components are displayed as change decriptions and the ability to trace back through the editing history of views and base components. ISDE developers using MViews have found the framework concepts simple to understand and straightforward to reuse.

VIII. Comparison to Other Approaches

Recent ISDE research has been concerned with abstract specification of software system structure and semantics, providing integrated textual and graphical views, managing the trade-off between integration and extensibility, and supporting collaborative software development.

A. Generated ISDEs

Declarative specification and generation of language-based environments has usually been based on an abstract syntax, together with attribute grammars for specifying semantic information. Examples are the Cornell Program Synthesizer (Reps and Teitelbaum 1987), MELD (Kaiser and Garlan 1987), and Mjølner environments (Magnusson et al. 1990). These environments are, however, only text-based. LOGGIE (Backlund et al. 1990), Dora (Ratcliffe et al. 1992), PECAN (Reiss 1985), and GLIDE (Kleyn and Browne 1993) use structure editing of views of shared, graph-based program representations, and graphical and textual view consistency is maintained by propagating editing changes between views. Weaknesses are that some change propagation can not be supported, such as some changes to design views which affect code views, and restrictive structure-editing is always used (Welsh et al. 1991).

Environments can be specified and generated more quickly using these formal grammar approaches than by framework specialisation. However, the usefulness and scalability of Dora, PECAN and GLIDE environments appears considerably less than that of SPE and EPE. These environments have less flexible view consistency mechanisms, as they use uni-directional constraint propagation rather than change descriptions. MViews also supports both fine-grained and coarse-grained storage of software system data making its data storage and performance more efficient. Mjolner grammars can be modified by users to support, for example, a different concrete syntax, whereas MViews environment classes must be modified to achieve this effect. Thus more work must be done to adapt MViews environments to individual developer's requirements. Also, experience with MViews has shown it takes new ISDE implementers longer to understand how to reuse the MViews framework than to use environment generators.

B. Framework ISDEs

A framework approach, as used by MViews, is less abstract, and generally involves more effort, than grammar-based ISDE generation. However, frameworks provide both a reusable model and also greater flexibility, as they provide general-purpose programming languages as well as specific abstractions for ISDE construction. With frameworks, environment implementers can code data management and user interface mechanisms differing in style from those envisaged by a framework's designers. In contrast, generated environments are restricted to capabilities offered by the generator language, usually less powerful than framework programming languages.

FIELD environments (Reiss 1990b) give the appearance of tight integration with extensibility via selective broadcast of events between Unix tools. However, building “wrappers” around tools to integrate them requires much work and the definition of broadcast events is complex and ad hoc (Meyers 1991). Garden is an environment for conceptual programming and for prototyping visual languages (Reiss 1987). It has limited support for multiuser software development via a shared repository based on an object-oriented database, but has no support for textual view consistency.

Unidraw (Vlissides and Linton 1990) is a framework for building domain-specific graphical editors, supporting multiple graphical (not textual) views. Rendezvous (Hill et al. 1994) and Garnet (Meyers 1991) are frameworks for building constraint-based graphical editors and user interfaces. Garnet uses unidirectional constraints. Rendezvous uses bidirectional constraints and also supports multiuser editing. Neither supports the flexible graphical or textual view consistency MViews provides.

FormsVBT (Avrahami et al. 1990), built using the Zeus framework (Brown 1991), supports interactive editing of graphical views and free-editing of textual views. View consistency is via token substitution in textual views and incremental redisplay of graphical views. Graphical view updates must be locked out when a textual view is edited, however, and only a simple S-expression language can be supported for the textual “program”. MViewsDP has similar functionality to FormsVBT, but MViewsDP's textual view consistency is more powerful, as information which does not overlap with graphical view information can be represented.

Escalante (McWhirter and Nutt 1994) supports both generation (via the GrandView environment) and framework specialisation approaches to visual environment construction. However, Escalante environments are small, single-user systems, compared to complex MViews environments such as SPE, EPE, Serendipity and ViTABaL.

C. Collaborative ISDEs

Many collaborative environments and CASE tools only support low-level editing mechanisms (Aean et al. 1992), including most Groupware systems (Ellis et al. 1991), Mercury (Kaiser et al. 1987), and Mjølner (Magnusson et al. 1993). Unlike Serendipity, these systems neither facilitate coordination of work, nor capture and presentation of work context information. Thus effective collaborative work on large systems is impossible. Some groupware systems support limited group awareness capabilities, such as multiple cursors (Roseman and Greenberg 1992), but these usually only inform collaborators about the work artefacts collaborators are immediately interested in and do not provide the context of others' work.

Process-centred environments utilise information about software processes to enforce or guide development. Examples include Marvel (Barghouti 1992), CPCE (Lonchamp et al. 1995), and ConversationBuilder (Kaplan et al. 1992) Computer-Aided Method Engineering (CAME) tools, such as Decamerone (Harmsen and Brinkkemper 1995) and Method Base (Saeki et al. 1993), provide support for configuring development processes and tools to a particular application. These approaches usually provide low-level text-based descriptions of work rationale, and often do not effectively handle restructuring of development processes while in use (Swenson 1993).

Table 1

Environment	Software Component Representation	Constraint Representation	Multiple Views	Consistency Management	Tool Integration	Collaborative Work Support	How New ISDEs are Built
MViews	CPRGs	CPRGs	Yes - via CPRGs	Change Description Propagation	Inter-repository rels.	Async., sync. editing; Serendipity	Toolkit (reuse Snart classes)
CPS	Abstract syntax trees	Attribute grammars	No	Attribute recalculation	Limited	None	Generated from grammar
GLIDE	Abstract syntax graph	?	Limited	?	Limited	Limited	Generated
FIELD	Unix files	?	Limited	message passing	message passing server	Limited	Coded in C
Rendezvous	Objects	Constraints	Yes - via ALV model	constraint propagation	Limited	Async., sync. editing	Coded
Zeus	Objects	?	Yes	Event propagation	Event propagation	Limited	Coded in Modula-3
Escalante	Objects	Simple constraint expressions	Yes	MVC-style	Limited	None	Partial generation
Mjølner	Abstract syntax trees	Object-oriented attributes	No	Attribute recalculation	backbone architecture	Async., sync. editing	Generated from grammars
Conversation Builder	Objects	?	Limited	Event propagation	Via event propagation	Async., sync. editing, workflow	Coded
Marvel/Oz	Rule-based system data	Rules on data	Limited	Rule application	via rules, database	concurrent transactions, work coordination	Rule-based system languages
Decamerone	Method fragments	Rules	Yes	Rule application	Yes	Limited	MEL language

Workflow-based systems, such as Active Workflow (Medina-Mora et al. 1992) and Domino (Kreifelts et al. 1991), attempt to coordinate work by describing the flow of documents between collaborators. This approach has proven to be inadequate for most real-world coordination activities. Exceptions to the workflows usually outnumber cases when they are useful, and the workflows often need to be modified while in use (Swenson 1993). Such systems usually do not model nor facilitate collaboration on the coordination (planning) activity itself (Swenson 1993). Serendipity provides high-level software process models for MViews ISDEs, and supports flexible event handling using an abstract visual language. Unlike most process-centred environments and workflow systems, Serendipity is tightly integrated with other MViews ISDEs.

D. Summarised Comparison

Table 1 shows a comparison of the facilities of the MViews toolkit for building ISDEs to a variety of other toolkits and environment generators. Being a toolkit, MViews enables ISDE developers to build more complex, flexible environments, but takes more effort to use than ISDE generator approaches.

IX. Summary

MViews provides a new approach to constructing integrated, extensible ISDEs with multiple textual and graphical views of software development. MViews represents software system data and multiple views of this data as dependency graphs. Discrete change descriptions to graph components are propagated to related components which then respond to these changes to maintain consistency. This mechanism supports environment integration and extensibility, as views (tools) share a common base data representation and new views and base components can be added without affecting existing components. Integrated, multiple textual and graphical views of software development are supported with bi-directional consistency management. A generic undo/redo facility, efficient incremental attribute recalculation, lazy consistency management, version control and collaborative software development facilities are also supported. Reuse of the object-oriented MViews framework allows flexible, practical ISDEs to be quickly constructed and maintained.

The MViews framework is being extended to provide more abstract support for attribute dependency specification and recalculation, specification of change description composition, and support for improved non-sequential undo/redo for version control. Generation of framework classes from a more abstract visual and textual specification of MViews environments is planned, which will allow further specialisation of generated classes, combining the advantage of a framework with the advantages of (partial) ISDE generation (Grundy and Venable 1996).

References

Aean, I., Siltanen, A., Sørensen, C., and Tahvanainen, V.P. (1992). "A Tale of Two Countries: CASE experiences and expectations." In *Proceedings of the IFIP WG8.2. Working Conference on The Impact of Computer Supported Technologies on Information Systems Development* (Minneapolis, June 14-17), Kendall, K.E., DeGross, J.I., and Lyytinen, K. Eds, North-Holland.

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

Arefi, F., Hughes, C.E., and Workman, D.A. (1990). "Automatically Generating Visual Syntax-Directed Editors," Communications of the ACM, vol. 33, no. 3, pp. 349-360.

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

Backlund, B., Hagsand, O., and Pherson, B. (1990). "Generation of Visual Language-oriented Design Environments," Journal of Visual Languages and Computing , vol. 1, no. 4, pp. 333-354.

Barghouti, N.S. (1992). "Supporting Cooperation in the Marvel Process-Centred SDE," in *Proceedings of the 1992 ACM Symposium on Software Development Environments* (Virginia, USA, December 9-11), ACM Press, pp. 21-31.

Bounab, M. and Godart, C. (1995). "A Federated Approach to Tool Integration," In *Proceedings of CAiSE'95* (Finland, June 13-16), Lecture Notes in Computer Science 932, Springer-Verlag, pp. 269-282.

Brown, M.H. (1991). "Zeus: A System for Algorithm Animation and Multi-View Editing," In *Proceedings of the 1991 IEEE Symposium on Visual Languages* (Kobe, Japan, Oct 9-11), IEEE Computer Society Press, pp. 4-9.

Dart, S.A., R.J., E., Feiler, P.H., and Habermann, A.N. (1987) "Software Development Environments," COMPUTER, vol. 20, no. 11, pp. 18-27.

Ellis, C.A., Gibbs, S.J., and Rein, G.L. (1991). "Groupware: Some Issues and Experiences," Communications of the ACM, vol. 34, no. 1, pp. 38-58.

Fenwick, S., Hosking, J.G., and Mugridge, W.B. (1994). "Visual debugging of object-oriented systems," In *Technology of object-oriented languages and systems TOOLS 15* (Melbourne, November), Prentice Hall*, pp. 9-19*

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

Grundy, J.C., and Hosking, J.G. (1995a). "Support for Integrated Formal Software Development," In *Proceedings of the 1995 Asia-Pacific Conference on Software Engineering* (Brisbane, Australia, Dec 6-9), IEEE CS Press, pp. 264-273.

Grundy, J.C. and Hosking, J.G. (1995b) "ViTABaL: A Visual Language Supporting Design By Tool Abstraction," In *Proceedings of the 1995 IEEE Symposium on Visual Languages* (Darmsdart, Germany, September 5-9), IEEE CS Press, pp. 53-60.

Grundy, J.C. and Venable, J.R. (1995). "Providing Integrated Support for Multiple Development Notations," In *Proceedings of CAiSE'95* (Finland, June 19-24)*,* Lecture Notes in Computer Science 932, Springer-Verlag, pp. 255-268.

Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. (1995a). "Connecting the pieces", Chapter 11 in*Visual Object-Oriented Programming*, Burnett, M., Goldberg, A., and Lewis, T. Eds, Manning/Prentice-Hall, Greenwich, Conneticut.

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

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

Grundy, J.C. and Hosking, J.G. (1996a). *Keeping textual and graphical views of information consisten*t, Working Paper, Department of Computer Science, University of Waikato.

Grundy, J.C. and Hosking, J.G. (1996b). *Serendipity: integrated environment support for process modelling, enactment and improvement*, Working Paper, Department of Computer Science, University of Waikato.

Grundy, J.C. and Venable, J.R. (1996) "Towards an environment supporting integrated Method Engineering," In *Proceedings of the IFIP TC8 WG8.1/8.2 Working Conference on Method Engineering* (Atlanta, August 26-28), Capman-Hall, pp. 45-62.

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

Grundy, J.C., Venable, J.R., Hosking, J.G., and Mugridge, W.B. (1996b) "Coordinating collaborative work in an integrated Information Systems engineering environment," in *Proceedings of the 7th Workshop on the Next Generation of CASE tools* (Crete, 20-21 May), Norwegian University of Science and Technology.

Harmsen, F., and Brinkkemper, S. (1995). "Design and Implementation of a Method Base Management System for a Situational CASE Environment," In *Proceedings of the 2nd Asia-Pacific Software Engineering Conference* (Brisbane, Australia, Dec 6-9), IEEE CS Press, pp. 430-438.

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

Hosking, J.G., Fenwick, S., Mugridge, W.B., and Grundy, J.C. (1995). "Cover yourself with Skin," in *Proceedings of OZCHI'95* (Nov 28-30, Wollongong, Australia, University of Wollongong Press, pp. 101-106.

Kaiser, G.E., Kaplan, S.M., and Micallef, J. (1987a). "Multiuser, Distributed Language-Based Environments," IEEE Software, vol. 4, no. pp. 11, 58-67.

Kaiser, G.E. and Garlan, D. (1987b). " Melding Software Systems from Reusable Blocks," IEEE Software, vol. 4, no. 4, pp. 17-24.

Kaplan, S.M., Tolone, W.J., Carroll, A.M., Bogia, D.P., and Bignoli, C. (1992) "Supporting Collaborative Software Development with ConversationBuilder," In *Proceedings of the 1992 ACM Symposium on Software Development Environments* (Virginia, USA, December 9-11), ACM Press, pp. 11-20.

Kleyn, M.F. and Browne, J.C. (1993). "A High Level Language for Specifying Graph Based Languages and their Programming Environments," In *Proceedings of the 1993 International Conference on Software Engineering,* IEEE CS Press, pp. 324-334.

Krant, R.E. and Streeter, L.A. (1995). "Coordination in Software Development," Communications of the ACM, vol. 38, no. 3, pp. 69-81.

Kreifelts, T., Hinrichs, E., and Klein, H.K. (1991). "Experiences with the Domino Office Procedure System," In *Proceedings of the Second European Conference on Computer Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam, pp. 117-130.

Lonchamp, J. (1995). "CPCE: A Kernel for Building Flexible Collaborative Process-Centred Environments," In *Proceedings of the 7th Conference on Software Engineering Environments* (Noordwijkerhout, Netherlands, April 5-7), IEEE CS Press, pp. 95-105.

Lyons, P., Simmons, C., and Apperley, M. (1993). "HyperPascal: Using visual programming to model the idea space," In *Proceedings of the 13th New Zealand Computer Society Conference* (Auckland, New Zealand, August 1993), Auckland University Press, pp. 492-508.

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

Magnusson, B., Asklund, U., and Minör, S. (1993). "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), ACM Press, pp. 7-10.

McWhirter, J.D. and Nutt, G.J. (1994). “Escalante: An Environment for the Rapid Construction of Visual Language Applications,” In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, IEEE CS Press, pp. 15-22.

Medina-Mora, R., Winograd, T., Flores, R., and Flores, F. (1992). “The Action Workflow Approach to Workflow Management Technology,” In *Proceedings of CSCW'92* (Toronto, Canada, Oct 31-Nov 4), ACM Press, pp. 281-288.

Meyers, S. (1991). “Difficulties in Integrating Multiview Editing Environments,” IEEE Software, vol. 8, no. 1, pp. 49-57.

Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. (1992). “Dora - a structure oriented environment generator,” IEE Software Engineering Journal, vol. 7, no. 3, pp. 184-190.

Reiss, S.P. (1985). “PECAN: Program Development Systems that Support Multiple Views,” IEEE Transactions on Software Engineering, vol. 11, no. 3, pp. 276-285.

Reiss, S.P. (1987). “Working in the GARDEN Environment for Conceptual Programming,” IEEE Software, vol. 4, no. 11, pp. 16-26.

Reiss, S.P. (1990a). “Interacting with the Field environment,” Software – practice & Experience, vol. 20, no. S1, pp. S1/89-S1/115.

Reiss, S.P. (1990b). “Connecting Tools Using Message Passing in the Field Environment,” IEEE Software, vol. 7, no. 7, pp. 57-66.

Reps, T. and Teitelbaum, T. (1987). “Language Processing in Program Editors,” COMPUTER, vol. 20, no. 11, pp. 29-40.

Roseman, M. and Greenberg, S. (1996). “Building Real Time Groupware with GroupKit, A Groupware Toolkit” , ACM Transactions on Computer-Human Interaction, vol. 3, no. 1, pp. 1-37.

Saeki, M., Iguchi, K., and Wen-yin, K. (1993). “A Meta-model for representing software specification and design methods,” in *Proceedings of the IFIP WG8.1 Conference on Information Systems Development* (Como, Italy), Prakash, N., Rolland, C., and Pernici, B. Eds.

Swenson, K.D. (1993). “A Visual Language to Describe Collaborative Work,” in *Proceedings of the 1993 IEEE Symposium on Visual Languages* (Bergen, Norway, August 24-27), IEEE CS Press, pp. 298-303.

Thomas, I. and Nejmeh, B. (1992). “Definitions of tool integration for environments,” IEEE Software, vol. 9, no. 3, pp. 29-35.

Venable, J.R. and Grundy, J.C. (1995). “Integrating and Supporting Entity Relationship and Object Role Models,” In *Proceedings of the 14th Object-Oriented and Entity Relationship Modelling Conferece* (Gold Coast, Australia, December 13-16), Lecture Notes in Computer Science 1021, Springer-Verlag, pp. 318-328.

Vlissides, J.M. and Linton, M. (1990). “Unidraw: A framework for building domain-specific graphical editors,” In ACM Transactions on Information Systems, vol. 8, no. 3, pp. 237-268.

Welsh, J., Broom, B., and Kiong, D. (1991). “A Design Rationale for a Language-based Editor,” *Software - Practice and Experience*, vol. 21, no. 9, pp. 923-948.

Acknowledgements

The authors gratefully acknowledge the helpful comments of the anonymous reviewers on an earlier draft of this paper. We also thank our colleagues Rick Mugridge and John Venable for helpful comments on various aspects of this work.