Revised version in the New Zealand Journal of Computing, Vol. 4, No. 2, 1993, pp. 31-40.

# The MViews framework for constructing multi-view editing environments

**JOHN C. GRUNDY and JOHN G. HOSKING**
*Department of Computer Science, University of Auckland*

ABSTRACT
MViews attempts to abstract out the common features of multi-view editing environments that support integrated textual and graphical programming with consistency management. It provides a conceptual model and reusable object-oriented framework for constructing interactive programming environments that provide multiple textual and graphical views. It supports multiple views of a base document, maintaining consistency between each of the views using an update record mechanism. MViews has been used to construct a visual programming environment for an object-oriented language featuring both graphical and textual views of the program. Other applications of MViews under development include entity-relationship and dataflow diagrammers, a visual debugger and a dialog box painter.

## 1. Introduction

Diagrams are useful in all phases of the software lifecycle to help explain and understand concepts that are difficult to describe in text (Davis, 1988). In object-oriented programming, for example, diagrams illustrating inheritance relationships are an invaluable aid in understanding program structure.

A natural extension of using diagrams to explain programs is to use diagram construction as a means of programming systems. This *visual programming* approach to program construction is becoming increasingly popular. Example visual programming systems include Fabrik (Ingalls et al, 1988), Prograph (Cox et al, 1990) and Garden (Reiss, 1986). Useful reviews of visual programming can be found in (Ambler and Burnett, 1989; Myers, 1990).

In previous work we have developed Ispel, a visual programming environment for object-oriented programming (Grundy et al, 1991). Ispel allows users to program either textually or graphically. In the latter mode, class structure diagrams can be constructed to define inheritance relationships and client-server relationships. An important feature of Ispel is its support of multiple views of a program. Multiple diagrams can be constructed with information shared between views. Modifications can be made to any of the views and the other views are automatically updated to be consistent.

In this paper we describe MViews, an abstraction from our earlier work on Ispel. MViews provides a framework to support the development of visual programming environments which include the multiple view with consistency model and free interchange between textual and graphical modes of programming. Visual programming environments for particular tasks, such as object-oriented programming or dataflow programming, are constructed by appropriately specialising MViews.

This paper begins with a description of the MViews conceptual model and architecture. This is followed by discussion of an object-oriented Prolog and its programming environment. This environment is a specialisation of IspelM, an MViews-based visual programming environment for object-oriented languages. An over-view of the implementation of MViews and IspelM is presented and the paper concludes with a discussion of current and future work.

## 2. MViews and IspelM Architectures

### Components of an MViews Environment

The central part of MViews is the program representation database which holds all information relating to program structure and different views of a program. Tools communicate via this central data repository which can also provide tool-specific data storage. Tools for a specific environment, such as text or graphic editors, are either tailor-made for the application or specialised from generic tools included in the MViews framework.

Graphical editors are structure-oriented, providing tools for manipulating specific aspects of a program, and utilise direct manipulation of graphical structures. Textual editors consist of an editor, an unparser and a parser. Unparsers convert a shared program representation into a textual form and parsers convert an edited piece of code into changes to this program representation. Existing language compilers and run-time systems can be used or new ones built using the MViews framework.

### Multiple View Support

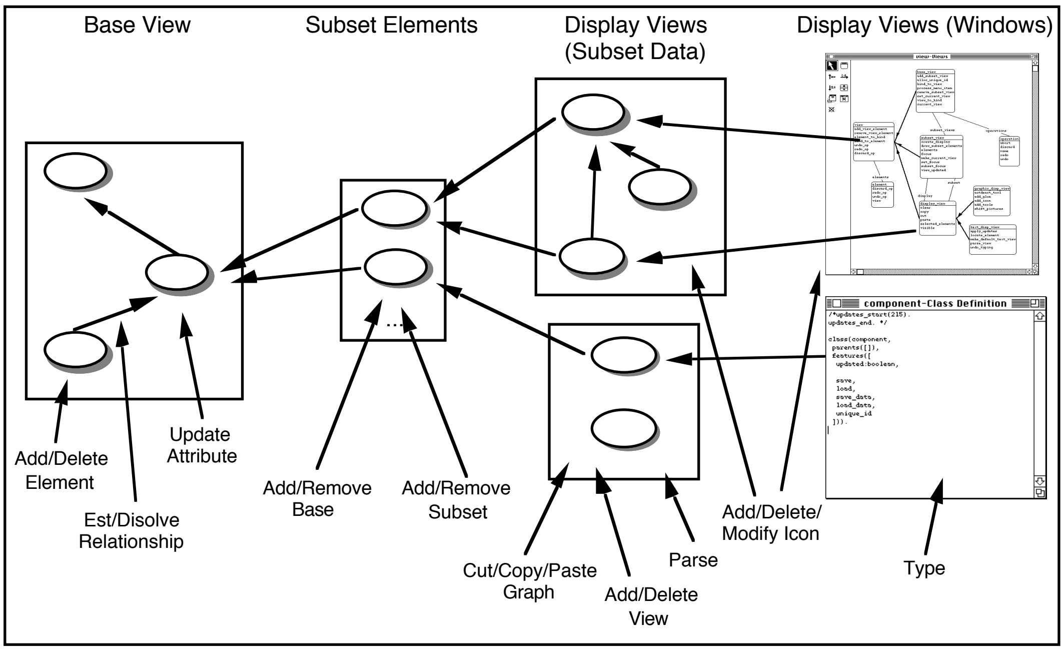
The main characteristic of MViews is its central support of multiple views. The term "multiple views" is used to describe related, yet distinct, ideas by different researchers. In MViews we define three types of view:

- *Base view:* This is a canonical representation of a complete program constructed as a synthesis of all other views. There is one base view per program.
- *Subset views:* These describe subsets of a base view and may overlap so the same information can be accessed and manipulated via different subset views. Examples of systems incorporating a similar notion to subset views include:
    - Ispel (Grundy et al, 1991), where multiple views describe overlapping subsets of a base view of an object-oriented program.
    - The dynamic and static views of MELD (Garlan, 1987) which partition programs into respectively overlapping and non-overlapping subsets.
    - Database views which filter out unwanted information. Database views are usually non-updatable, limiting the consistency management problems (although see (Langerak, 1990; Horwitz and Teitelbaum, 1986)).
- *Display views:* These describe how some part of the program is to be rendered on the screen and interacted with. The same program fragment can be rendered in a variety of notations, textual and graphical, using different display views. Many visual programming systems utilise some form of multiple display views. Examples include PICT (Glinert and Tanimoto, 1985), PECAN (Reiss, 1985), Garden (Reiss, 1986), and Ispel. Users interact with display views to modify either graphical

figures and connectors or textual characters which are translated into subset and base view modifications.

## Conceptual Foundations

MViews represents programs and views as collections of directed graphs. Thus program structure in MViews is specified in terms of program *elements* (graph nodes) and *relationships* between elements (labelled graph edges). Language semantic information for a particular program can be stored in the environment in an analogous manner. This program representation is similar to that employed by deterministic graph transformation systems (Arefi et al, 1990). Subset views of this program are constructed which themselves are directed graphs representing a subset of the base program graphs (with the components of these view graphs being a subset of base program components). Each subset is rendered (displayed) either graphically or textually. Hence a "view" of the base program is a subset/display pair which render several base program components.



```
/*updates_start(215).
updates_end. */

class(component,
 parents([]),
 features([
  updated:boolean,

  save,
  load,
  save_data,
  load_data,
  unique_id
 ])).
```

Figure 1. Some basic operations on MViews program graphs.

Graph *operations* are employed to modify a program graph. The semantics of these operations could be described as the editing semantics of the programming environment, i.e. the effect on the program state of applying an operation. Figure 1 shows typical operations affecting different view types and inter-view relationships.

Base view components store information about the total program graph. Display components store a subset of their base component data and render it in a textual or graphical form. Subset components relate base components to their display components. Subset components translate base updates into display updates and vice-versa. A base component need have no knowledge about its displays, and similarly a display no direct knowledge about its base, as subsets moderate all base/display updates. A display view may contain one or more disjoint graphs and display components need not be connected (mapped) to a base component.

A subset component may have several display components in the same or different display views and may have more than one base component. A base component may have several subsets (typically one for each “kind” of display) and a display can have several subsets. Thus MViews supports a very flexible base to display mapping with one display possibly rendering a subset of several base elements. Figure 1 also shows some typical base/subset/display components and their relationships.

## Propagation and Recording of Change

Propagation of change is an essential aspect of MViews’ multiple views. If shared information is modified in one view, a consistency management algorithm propagates the modifications to other views.

A change is typically made at the display level: text is modified and then parsed or graphical elements are added and updated. The display operations are then translated into base operations by subset elements. The base propagates this change to all its subsets which in turn update their displays.

MViews uses a concept of *update records* to propagate graph modifications and document program change. When a component is modified by an operation it “records” the change as an update record against itself. Base components store an update record to document changes they have undergone. Display components send their updates to their enclosing view which uses them to implement an undo/redo facility and partial “history” of their changes. As changes are made via display views, undoing a change is accomplished by sending its update record to the modified element for reversal (or re-execution for redo).

Every component has zero or more related components that may be affected by a change to itself (called *dependent components*). In addition to recording changes, components send their update records to these dependent components. Dependents interpret the update and modify themselves (if necessary), possibly generating further update records. This provides a mechanism for implementing inter-element constraints, data-driven programming and view updating. A base components’s dependents includes all of its subsets, a subset includes its base(s) and displays, and a display includes its subsets.

As change is sequential, update records can be applied by dependents immediately (data-driven), when the dependent requires a changed value (demand-driven), or a combination of both. This allows increased efficiency and provides a change propagation and view updating mechanism with flexibility similar to Hudson’s algorithm for attribute grammars (Hudson, 1991). As dependents are passed a complete update record, they do not have to reconcile themselves to their parent as views do in the Model-View approach of Smalltalk (Goldberg and Robson, 1984) and Subject-View of Unidraw (Vlissides, 1990). This also allows more flexible and efficient re-computation after change than Hudson’s approach (which only indicates that a parent attribute has changed and not how it has changed). Our program graph approach provides a similar level of abstraction to the list representation of (Dannenburg, 1990) but allows a more natural description program structure and comparable abstract update, undo and incremental redisplay capabilities using update records.

## IspelM Architecture

IspelM is an Ispel-like object-oriented programming environment which supports multiple textual and graphical views of an object-oriented program. An object-oriented program can be represented as an MViews graph with class and feature elements and generalisation and client-server relationships. A class diagram view of this program contains class icons (possibly including feature names) and generalisation and client-server icon connections. Textual views of a class include the class interface, its method implementations and documentation of class components.

## 3. The Snart Programming Environment

The first application of MViews has been in the development of a visual programming environment for an object-oriented Prolog called Snart. This involved a two step specialisation of MViews. The first step was to generate IspelM and further specialisation tailored IspelM for programming in Snart producing the Snart Programming Environment (SPE).

```
% Definition of the rectangle class
class(rectangle,
  parents([closed_figure([
    rename(create,fig_create),
    rename(info,closed_info)])
  ]),
  features([
    height:int,
    width:int,
    create, area,
    resize, draw,
    perimeter, info])).

% Create a rectangle
rectangle::create(Rect,Window,
    Location,Width,Height) :-
  Rect@width:=Width,
  Rect@height:=Height,
  Rect@fig_create(Window,Location).

% Area for a rectangle
rectangle::area(Rect,Area) :-
  Area is Rect@width * Rect@height.

% Resize a rectangle
rectangle::resize(Rect,NewX,NewY) :-
  Rect@width:=NewX,
  Rect@height:=NewY,
  Rect@draw.
```

```
% Draw a rectangle
rectangle::draw(Rect) :-
  Rect@window(Window),
  Rect@location((X,Y)),
  Rect@width(W),
  Rect@height(H),
  ( Rect@visible(true) ->
    Window@chg_pic(Rect,
      box(Y,X,H,W))
  ; Window@add_pic(Rect,
      box(Y,X,H,W))
  ),
  Rect@visible:=true,
  Rect@frame:=box(Y,X,H,W).

% Perimeter for a rectangle
rectangle::perimeter(Rect,Perim) :-
  Perim is
    2 * (Rect@width + Rect@height).

% Info for rectangle
rectangle::info(Rect) :-
  writenl('Info for rectangle:'),
  Rect@closed_info.
```

Figure 2. An example Snart class defining rectangles.

### Snart

Snart is an object-oriented extension to Prolog developed by the authors. We had previously used Prolog to good advantage in the development of Ispel (Grundy et al, 1991) but found the lack of structuring beyond the predicate level a disadvantage. Snart retains the advantages of Prolog programming but embedded within an object-oriented framework similar to that of Prolog++ (Pountain, 1990), ObjVProlog (Malenfant et al, 1988), and ProTALK (Quintus, 1992). Figure 2 shows an example of Snart code.

Snart provides class definitions which include attribute and method specifications. Multiple, repeated inheritance is supported together with redefining and renaming of features to avoid name clashes. Method predicates are defined separately in a C++ style and may have multiple clauses. Attributes can be assigned to and are therefore impure Prolog. They provide a structured alternative to using the standard Prolog assertion and retraction facilities. Programmers can freely mix Snart code and standard Prolog code. The implementation of Snart has aimed for efficiency of execution with object creation, storage, method dispatch and attribute access optimized.

## An Overview of SPE

SPE supports multiple textual and graphical views of a Snart program sharing common information. Multiple views allow the construction of many diagrams each focussing on a particular aspect of the program thus reducing the cognitive complexity in understanding those aspects of the program. Consistency management is employed to maintain data integrity between all views, textual or graphical, that share information.

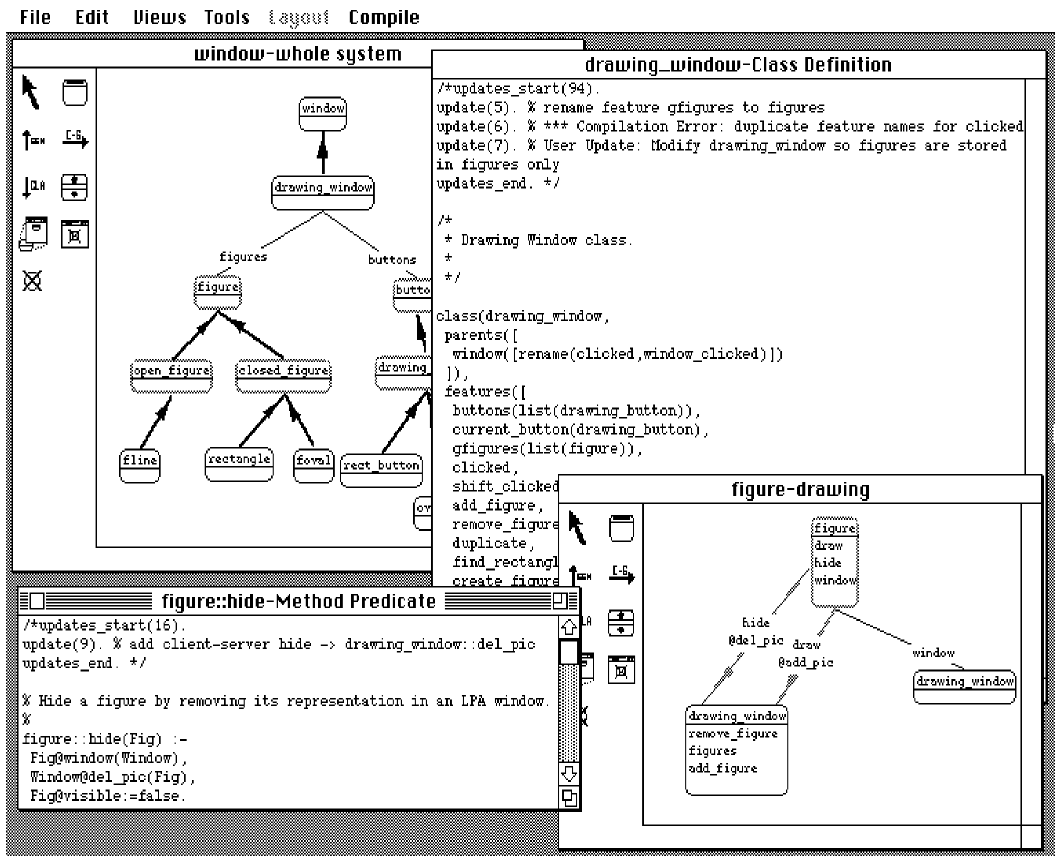
Each view supports the most appropriate method of manipulation. Graphical views are structure-edited using a palette of supplied tools and menus while textual views are free-edited and parsed. This differs from comparable approaches that only employ restrictive structure-oriented editing (Minör, 1990; Ratcliff et al, 1992).

Figure 3 shows SPE editing a simple drawing program implemented in Snart. One graphical view shows the `button` and `figure` inheritance hierarchies. The other shows client-server relationships between the `drawing_window` and `figure` classes in the context of adding and rendering pictures in a window. One textual view shows the class definition for `drawing_window` and the other the `hide` method predicate for `figure`.

Programmers typically use graphical views for analysis and design of their programs and to visually document a program to enhance readability and browsing. These views provide class icons with feature names (including inherited features), abstract and concrete classes, generalisation, association and client-server relationships, and classification. Tools and menus are used to interactively add, update, and remove icons and connectors.

Textual views are typically used to implement method predicates, to add Prolog predicates, and to specify additional class definition details such as renaming of inherited features. Arbitrary documentation of parts of a program describing design or implementation detail can be added using textual views. After parsing textual views the existing Snart compiler is then used to generate Snart code. Programs can be run and debugged using the Prolog run-time system and object data is displayed using graphical object views.

Programmers determine the feature names displayed in a class icon and the relationships shown in a graphical view. Textual views may contain several class definitions or method predicates at a programmer's discretion. Class definitions may be canonical (representing a Snart class) or a class "view" with only some of a class's total features (including inherited features) being displayed. Documentation and Snart code textual forms may be freely mixed within a view. These facilities assist programmers in managing the complexity of their programs by showing detail only of relevance to the view.



```
/*updates_start(94).
update(5). % rename feature gfigures to figures
update(6). % *** Compilation Error: duplicate feature names for clicked
update(7). % User Update: Modify drawing_window so figures are stored
in figures only
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_rectangl
  create_figure
```

```
/*updates_start(16).
update(9). % add client-server hide -> drawing_window::del_pic
updates_end. */

% Hide a figure by removing its representation in an LPA window.
%
figure::hide(Fig) :-
 Fig@window(Window),
 Window@del_pic(Fig),
 Fig@visible:=false.
```

Figure 3. The Snart Programming Environment.

Programmers need to locate information easily from a large number of views and be able to gain a high-level over-view of different program aspects. SPE's approach is to use the program views themselves in a hypertext-like fashion as the basis for browsing. Class icons in graphical views have a number of active regions, or "click-points", which cause pre-determined actions to be carried out when clicked on, similar to Prograph's dataflow entities (Cox et al 90). Click points allow rapid navigation to any other views (textual or graphical) containing the class or individual features of the class. SPE provides menus for textual views that perform similar facilities to click points.

Programmers can construct additional views for the sole purpose of program browsing, based on information selected from other views via dialogs. This provides a very flexible static program visualisation mechanism with view composition and layout under the complete control of a programmer. A caching system means not all views and base data need be in memory at once; updates can be propagated to views when they are selected and reloaded and base data can be incrementally loaded and saved as required.

## Managing Change in SPE

Object-oriented software development tends to be an evolutionary process (Henderson-Sellers and Edwards, 1990; Coad and Yourdon, 1991). Hence program design and implementation may require change for a variety of reasons. A design may be incomplete and require modification which impacts on its implementation. Errors in an implementation must be corrected which in turn may highlight deficiencies in a design. Changes may even be transient in that they inform programmers of tasks to perform or errors requiring correction. Many CASE tools and programming environments provide facilities for generating code based on a design (Coad and Yourdon, 1991; Wasserman and Pircher, 1987) but few provide consistency management when code or design are changed.

A key feature of SPE (and MViews) is the method of recording changes and handling updates to views that result from changes to another view. In some cases, such as a change to a feature name, updates can be made directly. In other cases, such as adding an abstract client-server connection, it is not possible to automatically infer the correct modification to a view and user assistance is needed. For this reason, updates to views are not always immediately performed. Rather, an indication of the impending update is given to the user who can then either accept, provide an implementation for, or reject the update.

As an example, a modification to a graphical view such as renaming the `gfigures` feature of `drawing_window` to `figures`, is reflected in a corresponding textual view by an update record as shown in Figure 3. This record informs the programmer of the change to the base data and allows her/him to either make the change or to select the update record and have SPE make the change to the text (in this case, changing `gfigures : list(figure)` to `figures : list(figure)`).

```
/*updates_start(94).
update(5). % rename feature gfigures to figures
update(6). % *** Compilation Error: duplicate feature names for clicked
update(7). % User Update: Modify drawing_window so figures are stored in
figures only
updates_end. */

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
  parents([
   window([rename(clicked,window_clicked)])
  ]),
  features([
   buttons:list(drawing_button),
   current_button:drawing_button,
   ...
```

Figure 4. Different types of updates for `drawing_window`.

Another update record is shown in the `figure::hide` view. A client-server link has been added between `drawing_window` and figure indicating that the `del_pic` feature of the `drawing_window` is used by the `hide` feature of `figure`. For this change automatic update of the textual view is not possible as SPE cannot infer the appropriate modification to the `hide` method and the user must implement the update.

Going from textual to graphical (or other textual) views is handled in a similar manner. After parsing a textual view, updates to the program are determined by changes to the corresponding parse tree. Any updates are reflected in graphical views by applying the change (for example, if a feature is renamed), or displaying the changed data in a different colour (for example, red for a deleted feature).

Updates are propagated to all multiple graphical and textual views sharing changed information.

Updates are also used to inform users of semantic or compilation errors (syntax errors are flagged interactively) and to document changes. Programmers can add arbitrary "user-defined" updates that describe various changes performed or to perform on a program. A compilation error and user-defined update are shown in the `drawing_window` class definition in Figure 4. The complete set of update records for a program component can be viewed in a dialog and updates either deleted or recorded as "history updates" (which document previous changes).

## 4. Design and Implementation

To produce a reusable MViews system either a programming environment (PE) generator with its own specification language similar to that of the Synthesizer Generator (Reps and Teitelbaum, 1984) could be constructed, or a specialisable framework implemented as in Unidraw (Vlissides, 1990). Many aspects of a good, interactive PE, such as the editor functionality and tool interfacing systems, require specialisation and fine-tuning on a scale difficult to provide with a specialised PE generator. Generated PEs are also well known for their poor user interfaces and performance (Minör, 1990). For these reasons the second approach was chosen.

Object-oriented languages foster reuse in various ways (Meyer, 1988) and a specialisable MViews framework lends itself to an object-oriented representation and implementation. Generalisations can be used to relate parts of the environment and specialisation and genericity allow reuse of these abstractions. Type aggregation and client-server relationships allow attributes and operations to be attached to appropriate classes and accessed and inherited via well-defined mechanisms.

### The MViews Framework

Development of the MViews architecture commenced with a denotational semantics specification of the graph representation of program state and the operations performed on that state (including a formal treatment of undo/redo operations). From this specification an object-oriented design and implementation followed. MViews class hierarchies were derived from the formal specification and used to structure the framework. Class responsibilities and services were then determined and classes implemented. MViews provides a collection of abstract classes that implement or provide a framework for MViews-based systems.

MViews components are defined as Snart classes, operations are implemented as Snart method calls and attributes as Snart object attribute values. All MViews components (views and view elements) support the basic operations of fetching and updating attributes, manipulating list attributes (for efficiency), recording and propagating updates, and component deletion.

Base program views hold all data about a program. Element classes hold attribute and relationship information with attributes implemented as Snart attributes. Relationships are modelled as elements, an element with a list of element references, or an element reference or list of element references. Operations are performed by a set of defined features (such as attribute update or element deletion) and can be augmented with more complex operations by sub-classing. As all operations are implemented as features, sub-classing can over-ride default implementations to provide language semantics and constraints support. Updates are recorded in an application-specific form against base elements.

Subset elements translate base updates to display updates and vice-versa. Subsets translate element creation and deletion and attribute updates generically. Sub-classing can extend this simple update propagation to provide arbitrary translation mechanisms. Base changes are propagated to subset elements and are applied to each of the subset's displays in the current (front) view. If a display is not in the current view, updates are cached against its view and the display element updated either incrementally or when its view becomes the current view.

Textual and graphical display views render subsets in either a graphical or textual form. They provide operations to manipulate display elements and views, and record display update records for undo/redo. Display elements are partial copies of base elements and provide features to perform standard operations, find a base element to map to when added, and generate their view rendering. Undo of display operations is completely generic for basic operations and is easily extended by sub-classing. Updates are passed from subsets to displays to re-render them (possibly incrementally, depending on the update) and displays send operations to their subsets which modify their base (if necessary).

Graphical views provide editors which are structure-oriented. They include tools and menus that act upon icons and connector glue to effect changes to display elements. The built-in MViews text editor is used to manipulate textual views but a standard text editor could also be used. The former provides menus to support view navigation, structured access to subset information and multiple text forms. Views are parsed to update MViews data and assert Prolog terms using either the Prolog parser or a user-defined parsing algorithm.

MViews provides generic routines to store and reload element and view data in a persistent form. This includes incremental saving and loading of both base and view data and supports storage schema evolution during development. The current facility uses Macintosh text resources and stores information as a collection of Prolog terms. While this provides a very flexible storage scheme, we plan to use a more versatile PCTE-like object database in future versions of MViews, similar to that employed by Dora (Ratcliffe et al, 1992).

MViews does not currently support version control or multi-user updates to shared programs. We are extending MViews so update records can be recorded in groups and undone and redone in an arbitrary order to support very flexible version control. We also plan to use update records to moderate con-current access and modification of programs.

## The IspelM Framework and SPE

IspelM is a specialisation of MViews and itself provides a framework for implementing programming environments for object-oriented languages. IspelM defines classes which are specialisations of MViews framework classes. These implement:

- Program representation and storage including program, cluster, class and feature elements, and generalisation, client-server and classifier relationships.
- Graphical display views for describing and manipulating class relationships and displaying run-time object data.

- Textual display views and forms for implementing class definitions and method predicates, documenting programs, and showing partial class information.

IspelM uses MViews to implement program storage and view construction facilities. IspelM defines specialised operations and renderings for graphical views, parse-tree processing for textual views, and unparsing and translation mechanisms for update records. MViews provides IspelM's view navigation and program persistency facilities, basic textual and graphical view manipulation tools, and a framework for supporting update records.

SPE specialises IspelM to support development of Snart programs. SPE includes Snart-specific parsers and unparsers for textual display views, and saving and loading support for Prolog code. Graphical views and base information require little change to support a different language as common object-oriented concepts are captured well at the IspelM level. An interface from IspelM to the language compiler and run-time system is necessary for different languages.

### Experience with SPE

MViews and SPE are currently implemented in Snart and run under LPA MacProlog on the Macintosh. The SPE environment performance is acceptable for designing, implementing, and debugging Snart programs although both graphical view interaction and program saving and loading can be improved. We have developed several small programs in SPE to illustrate the capabilities of the environment. SPE is also used as an object-oriented analysis and design tool in addition to an environment for the construction of Snart software.

## 5. Summary and Current and Future Research

We have described MViews, a framework for developing visual programming environments featuring multiple views with consistency management. MViews has been applied in the development of IspelM, a generic environment for object-oriented programming, and SPE, a specialisation of IspelM for visual programming in Snart. SPE provides a multiple view programming environment with both textual and graphical manipulation of Snart programs. Design-level and implementation-level changes are recorded and both phases of development kept consistent using a uniform model.

Other applications of MViews are currently under development. A dataflow programming tool, after the style of Prograph (Cox et al, 1990), will provide an object-oriented dataflow diagraming tool. These dataflow programs can be integrated with Snart code allowing a mixture of conventional and dataflow programming in SPE. A visual debugger for Snart programs uses a similar approach to the SPE graphical tools but displays the state of objects rather than classes. A dialog box "painter" is a visual tool for laying out dialog boxes which can then be included within a Snart program or with conventional Prolog programs. This may be extended to support more general user interface building and programming-by-example. An entity-relationship diagraming tool provides graphical entities and relationships which are translated into relational schema that are viewed and manipulated in a textual view.

Future applications we envisage for MViews include specialisations of IspelM for object-oriented languages other than Snart including Eiffel (Meyer, 1988) and Kea (Hosking et al, 1990). Further specialisation of IspelM for object-oriented analysis (Booch, 1991; Coad and Yourdon, 1991) would provide facilities more abstract than the current design-implementation-maintenance views of IspelM but should allow progressive refinement through to a full implementation. Change propagation between analysis and design could be supported using the update record model. Program visualisation tools will provide a more graphical and dynamic view of program execution than that provided by the visual debugger. Using MViews for both schema and instance browsing and manipulation in program visualisation and future versions of ICATect (Amor et al, 1992) will extend its program-based model. A multi-user environment with shared access to program data and views is envisaged with support for versioning and concurrent updates.

## Acknowledgements

The helpful comments of the anonymous reviewers greatly assisted production of this paper. The financial assistance of the Building Research Association of New Zealand and the University of Auckland Research Committee is gratefully acknowledged. John Grundy was supported by an IBM Postgraduate Scholarship, a William Georgetti Postgraduate Scholarship and a New Zealand Universities Postgraduate Scholarship while pursuing this research.

## References

Ambler, A., Burnett, M. (1989): Influence of Visual Technology on the Evolution of Language Environments, In *IEEE Computer,* **22** (9), 1989, 9-22

Amor, R.A., Hosking, J.G., Groves, L.J., Donn, M.R. (1992): Design tool integration: model flexibility for the building profession, In *Proceedings Building Systems Automation-Integration 1992 Symposium: Computer Integration of the Building Industry*, Dallas, Texas, 1992.

Arefi, F., Hughes, C.E., and Workman, D.A. (1990): Automatically Generating Visual Syntax-Directed Editors, In*CACM,* **33** (3), 1990, 349-360.

Booch, G. (1991): *Object-Oriented Design with Applications* Menlo Park, CA, Benjamin/Cummings, 1991.

Coad P., Yourdon, E. (1991): *Object-Oriented Analysis*, Second Edition, Yourdon Press, 1991.

Cox, P.T., Giles, F.R., Pietrzykowski T. (1990): Prograph: a step towards liberating programming from textual conditioning, In *Proceedings of 1990 IEEE Workshop on Visual Languages*, 1990, 150-156.

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

Davis, A.M. (1988): A Comparison of Techniques for the Specification of External System Behaviour', In*CACM*, **31** (9), 1988, 1098-1115.

Garlan, D. (1987):*Views for Tools in Integrated Environments*, PhD Thesis, Carnegie-Mellon University, CMU-CS-87-147, 1987.

Glinert, E.P., and Tanimoto, S.L. (1985): PICT: An interactive, graphical programming environment, In *IEEE Computer,* **17** (11), 1985, 7-25.

Goldberg, A. and Robson, D. (1984): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading MA., 1984.

Grundy, J.C., Hosking, J.G., and Hamer, J. (1991): A Visual Programming Environment for Object-Oriented Languages, In *Proc TOOLS 5,* Prentice-Hall, 1991, 129-138.

Henderson-Sellers, B. and Edwards, J.M. (1990): The Object-Oriented Systems Life Cycle, In *CACM*, **33** (9), 1990, 142-159.

Horwitz, S. and Teitelbaum, T. (1986): Generating Editing Environments Based on Relations and Attributes, In*ACM TOPLAS,* **8** (4), 1986, 577-608.

Hosking, J.G., Hamer, J., Mugridge, W.B. (1990): Integrating functional and object-oriented programming, In *Proceedings of TOOLS3,* TOOLS Pacific, Sydney, 1990, 345-355.

Hudson, S.E. (1991): Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, In *ACM TOPLAS* **13** (3), 1991, 315-341.

Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., Doyle, K. (1988): Fabrik: A Visual Programming Environment, In *Proc OOPSLA '88*, 1988, 176-189.

Langerak, R. (1990): View Updates in Relational Databases with an Independent Scheme, In *ACM Transactions on Database Systems,* **15** (1), 1990, 40-66.

Malenfant, J., Lapalme, G., and Vaucher, J. (1988): ObjVProlog: Metaclasses in Logic, In *Proceedings of ECOOP '89 Conference*, Cambridge University Press, 1989, pp. 257-269.

Meyer, B. (1988): *Object-Oriented Software Construction*, Prentice-Hall, 1988.

Minör, S. (1990): *On Structure-Oriented Editing*, PhD Thesis, Department of Computer Science, Lund University, Sweden, 1990.

Myers, B.A. (1990): Taxonomies of Visual Programming and Program Visualization, In *Journal Visual Languages and Computing*, **1** (1), 1990, 97-123.

Pountain, R. (1990): Adding objects to Prolog, In *Byte*, August (1990), 1990, 64IS-15 - 64IS-20.

Quintus Corporation (1992): *ProTALK 1.0 Reference Manual*, Quintus Corporation, 2100 Geng Road, Palo Alto, CA 94303, 1992.

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

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

Reiss, S.P. (1986): GARDEN Tools: Support for Graphical Programming, In *Lecture Notes in Computer Science #244*, Springer-Verlag, 1986, 59-72.

Reps, T. and Teitelbaum, T. (1984): The Synthesizer Generator, In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, New York,1984, 42-48.

Vlissides, J.M. (1990): *Generalized Graphical Object Editing*, PhD Thesis, Stanford University, CSL-TR-90-427, 1990.

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