TOOLS Pacific 1992 Conference, Sydney, Prentice-Hall, November, 1992, pp. 129-137.

MViews: A Framework for Developing Visual Programming Environments

J.C. Grundy and J.G. Hosking
Department of Computer Science
University of Auckland,
Auckland, New Zealand
kea@cs.aukuni.ac.nz

Abstract

MViews is a framework for constructing visual programming environments. It supports multiple views of a base document, maintaining consistency between each of the views. 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. 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 1988], Prograph [Cox 1990], and Pegasys [Moriconi 1986]. Useful reviews of visual programming can be found in [Ambler 1989] and [Myers 1990].

In previous work we have developed Ispel, a visual programming environment for object-oriented programming [Grundy 1991]. Ispel allows users to program either textually or graphically. In the latter, 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 overlapping information in each view. 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, a generalisation of Ispel. MViews is a programming environment framework. Visual programming environments for particular tasks, such as object-oriented programming or dataflow programming, are constructed by appropriately specialising MViews.

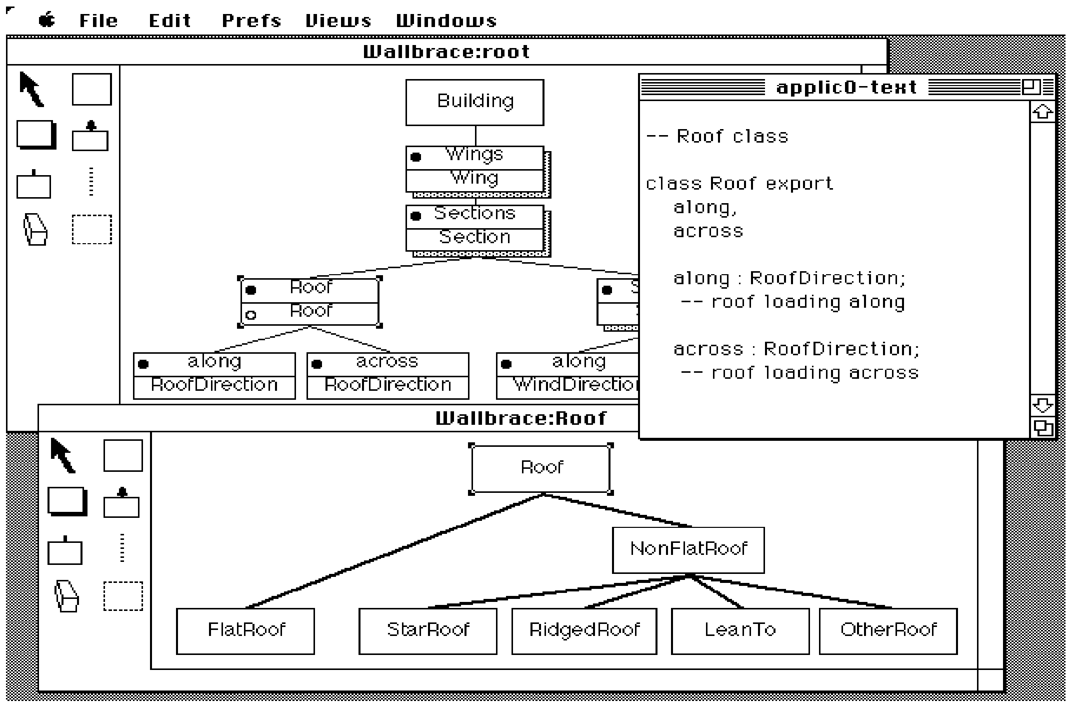
The paper begins with a brief review of Ispel, followed by an introduction to the MViews architecture. Implementation of MViews and its application in the development of an Ispel-like environment, IspelM, are then described. The paper concludes with a discussion of current and future work.

2. Ispel

Figure 1 shows Ispel in use, illustrating aspects of the programming environment and some of the types of view available. The following summarises significant features of Ispel:

- Multiple views of the program are supported. Each view may share information with other views. The program as a whole is the union of the information supplied by each view. Each view can occupy its own window or share one with other views. Three views are shown in Fig. 1, two graphical and one textual.

- Multiple view support allows diagrams focussing on particular aspects of the program to be constructed. This reduces the cognitive complexity in understanding those aspects of the program. For example one graphical view in Fig. 1 has been used to describe the inheritance relationship between different varieties of Roof classes, while the other shows the major feature hierarchy for Building objects.



```
-- Roof class

class Roof export
   along,
   across

   along : RoofDirection;
    -- roof loading along

   across : RoofDirection;
    -- roof loading across
```

Figure 1. Example of Ispel in use.

- Each graphical view includes a palette of tools used to construct and edit the view. Tools for constructing, removing, and expanding (based on information from other views) classes, inheritance links, and feature links are provided.

- Detailed code of a feature can be programmed using text. The applic0-text view in Fig. 1 shows a textual view of the Roof class. This may be edited to provide additional features, feature bodies, etc. This version of Ispel supports Eiffel [Meyer 1988] syntax. Another version supports programming in Kea [Hosking 1990; Hosking 1991].

- Any view can be modified. All other views that share affected information are updated to maintain consistency.

- Consistency management also applies between textual and graphical views. Modifying a graphical view may cause a textual view to be updated or vice-versa. Users are thus free to program in whatever mode, text or graphics, they feel more comfortable with, but can view and manipulate their results in views of the other mode.

- View management facilities allow views to be created, hidden, made visible, and removed. View navigation facilities provide a variety of ways of reaching appropriate views.

Ispel is designed as a tool for visually programming object-oriented systems. However, we recognised that some aspects of the Ispel environment could have wider application in the development of other visual programming environments, specifically:

- The multiple view with consistency model
- The free interchange between textual and graphical modes of programming

A framework supporting such facilities was therefore felt to be desirable. The provision of these facilities in Ispel is quite closely bound to the OO-specific facilities, so direct abstraction of such a framework from the existing Ispel implementation would have proved difficult. For this reason, MViews, a new framework, was developed.

3. MViews

Fig. 2 shows the major components of a programming environment constructed using MViews. Central 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.

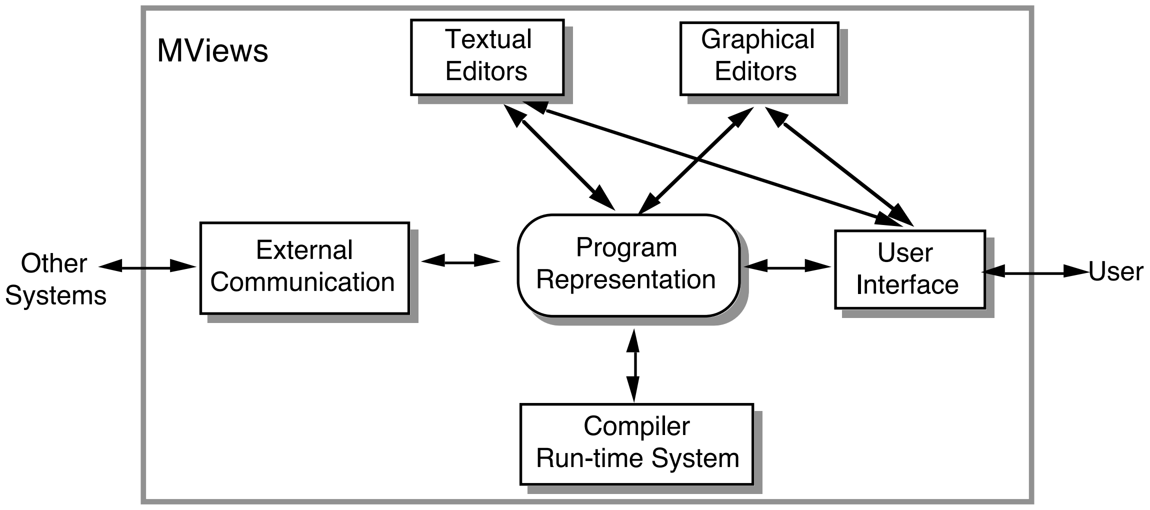


Figure 2. Components of an MViews-based programming environment.

Tools for a specific environment, such as text or graphical 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 the common repository format.

The main characteristic of MViews is its central support of multiple views, unlike most other systems which tend to treat views as an additional component of the programming environment. 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 a single base view for any one program.
- *subset views:* These describe subsets of a program. Subset views 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, 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, however, limiting the consistency management problems (although see [Horowitz 1986] and [Langerak 1990]).

- *display views:* These describe how some part of the program is to be rendered on the screen. 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 1984], PECAN [Reiss 1985], Garden [Reiss 1986], and Ispel. Most of these, though, only provide several ways of rendering a single base view. MViews' display views visually render a subset view, allowing only a specified part of the program to be displayed by the renderer. Users interact with display views to modify graphical figures and connectors, or textual characters. These modifications are translated into subset and base view program modifications.

Propagation of change is an essential aspect of MViews' multiple views. If shared information is modified in one view, a consistency manager propagates the modifications to other views. For example, modification of a display view may alter the base program state. Other views affected by this change must then be updated and redisplayed, to provide a consistent presentation of the program across the environment. Change propagation is controlled by each affected view: the view is notified of a change, it updates its components appropriately, and then propagates further changes to related views (and hence their components).

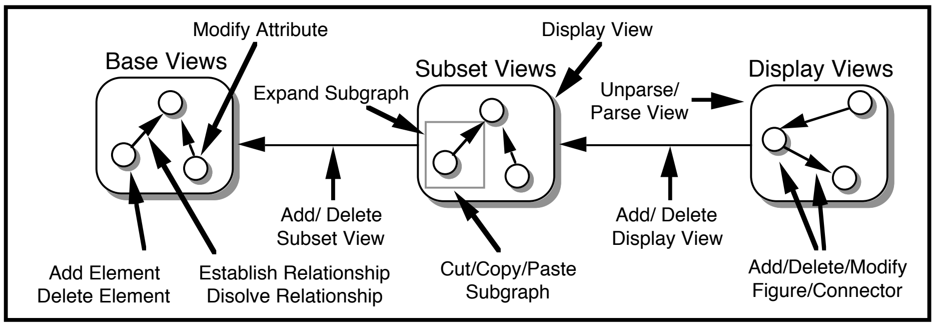


Figure 3. Some basic operations on MViews programs.

MViews represents programs and views as collections of directed, acyclic 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 1990].

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: the effect on the program state of applying an operation. Some basic operations include adding elements, establishing relationships between elements, deleting elements, dissolving relationships, and modifying the attributes of elements and relationships. Fig. 3 shows typical operations affecting the different view types and inter-view relationships.

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, as discussed in the next section.

4 Design and Implementation

To produce a reusable MViews system, a programming environment (PE) generator with its own specification language could be constructed, similar to that of the Synthesizer Generator [Reps 1984], or a specialisable framework implemented, as used in Unidraw [Vlissides 1990]. However, 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. Also, generated PEs are well known for their poor user interfaces and performance [Minör 1987]. 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 the client-server relationship allow attributes and operations to be attached to appropriate classes, and accessed and inherited via well-defined mechanisms. View membership determination, operation reversal and delaying, and tool interfacing and specialisation, are all suitable for object-oriented implementation.

In designing MViews, class hierarchies were derived from the formal specification, and used to structure the framework. Class responsibilities and services were then determined. Fig. 4 shows part of this framework, in terms of some of the main classes (boxes) and generalisations (arrows) used.

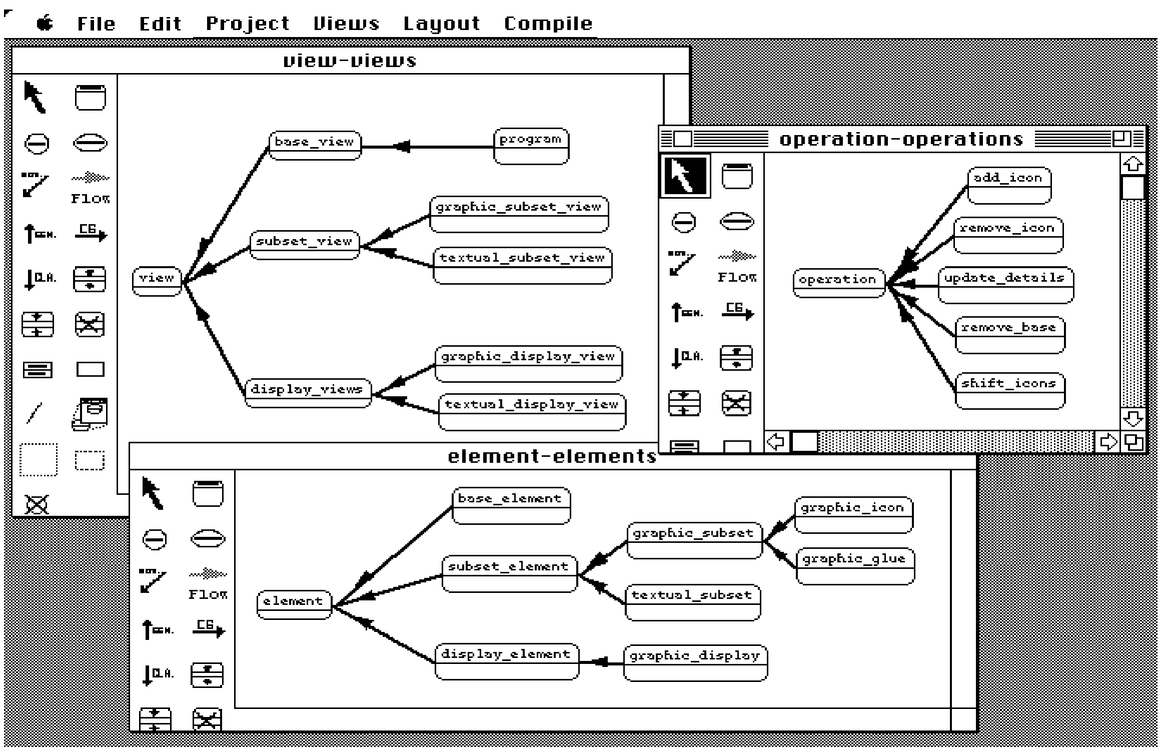


Figure 4. An object-oriented framework for MViews

```
class(rectangle,
 parents([ closed_figure(
     [rename(create,fig_create),
      rename(info,closed_info)])
     ]),
 attributes([ height(int),
             width(int)]),
 methods([ create, area,
           resize, draw,
           perimeter, info])).

% Create a rectangle
rectangle::create(Rect,Window,Loc,W,H)
:-
   Rect@width:=W,
   Rect@height:=H,
   Rect@fig_create(Window,Loc).

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

% Resize a rectangle
rectangle::resize(Rect,NX,NY) :-
  Rect@width:=NX,
  Rect@height:=NY,
  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,Perimeter)
:-
   Perimeter is
        2 * (Rect@width +
Rect@height).

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

Figure 5. An Example Snart class defining Rectangles

Implementation of MViews is in Snart, an object-oriented extension to Prolog developed by the authors. We had previously used Prolog to good advantage in the development of Ispel [Grundy 1991], but found the lack of structuring beyond the predicate level a disadvantage. Snart aims to retain the advantages of Prolog programming, but embedded within an object-oriented framework similar to that of [Pountain 1990].

Fig. 5 shows an example of Snart code. Snart includes the following features:

- Classes contain attribute and method specifications.
- Method predicates are defined separately in a C++ style. Method predicates may have multiple clauses.

- Creation methods are used to create and initialise an object

- Attributes can be assigned to and are therefore impure Prolog. They provide a structured alternative to using the standard Prolog assertion and retraction facilities.

- Multiple inheritance is provided, together with redefining and renaming of features to avoid name clashes.

- Programmers can freely mix Snart code and standard Prolog code
- The implementation of Snart has aimed for efficiency of execution. Snart code is compiled to Prolog. Object creation, storage, method despatch and attribute access have all been optimised.

MViews provides a collection of abstract classes that implement or provide a framework for:

- Storage of base program data describing program components. For example, classes, features, clusters, generalisations, client-server relationships, and program documentation for IspelM.

- Textual and graphical subset views of base data. These subset views are partial copies of the base, and can be modified by editors, or by changes to the base data.

- Change propagation to maintain consistency between the base view, graphical, and textual subset views. This includes demand- and data-driven view update algorithms, and visual notification of updates in both graphical and textual views.

- Textual and graphical display views that render subsets in either a graphical or textual representation. Both types of views may be edited by users to effect changes at the subset, and consequently, the base levels. Graphical views are structure-edited, while textual views are free-edited and then parsed.

- Graphical structure-editing and text editor facilities. Graphical editors include tools that act upon icons and connector glue to effect changes to subset views. A standard text editor can be used to manipulate textual views, or the built-in MViews text editor can be used. The latter provides hyper-text links to enable view navigation and structure-based searching.

- Operation storage for subset views that implement undo/redo facilities. Operation histories are also provided. These are completely generic, requiring no code be added to specialisations of MViews (such as IspelM) to implement undo/redo.

- Generic routines that save and reload MViews data to and from persistent storage. These include incremental saving and loading of both base and subset view data.

- Support for application-specific semantics processing. For example, in IspelM unique names must be used for classes, and for each feature per class, and these semantic constraints are added to classes implementing IspelM.
- Unparsing and parsing support for textual views, including parse-tree storage and determination of base view updates via parse-tree changes.

- An object-oriented interface to the Macintosh user-interface system, including window, dialog, menu, and editing tool support.

5. Application

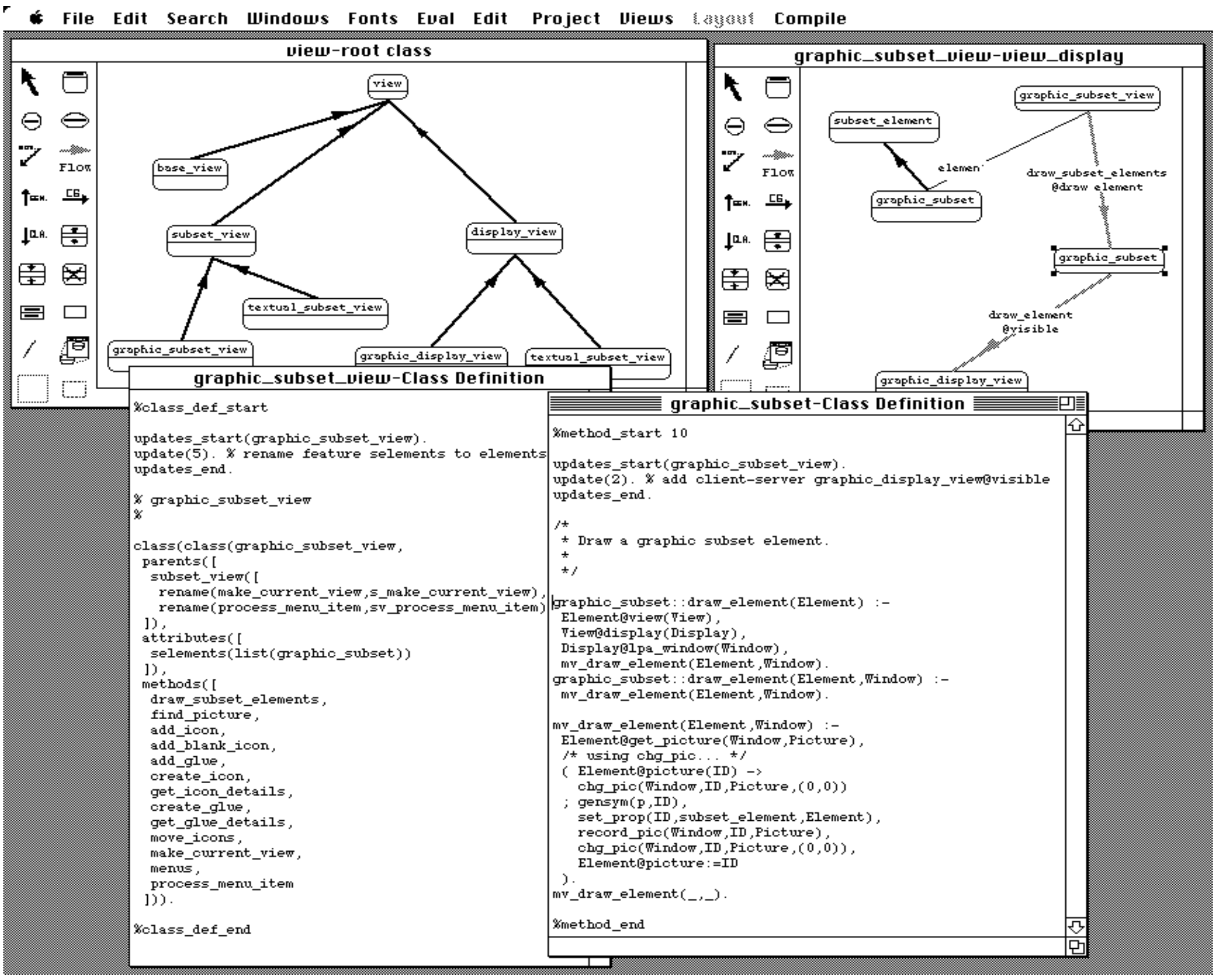
The first application of MViews has been in the development of a visual programming environment for Snart itself. This involved a two step specialisation of MViews. The first step was to generate an Ispel-like object-oriented programming environment in MViews (IspelM). Further specialisation tailored IspelM for programming in Snart producing the Snart programming environment (SPE).

Fig. 6 shows SPE in use. The environment provided is quite similar to that of Ispel, providing multiple graphical and textual views. However, SPE provides full graphical and textual view consistency (in both directions), and has a much richer set of visualisation capabilities and representations.

The example shows SPE editing itself (IspelM and MViews are both implemented in Snart). The graphical views show the view inheritance hierarchy, and client-server relationships between graphical subset and display views and their elements, in the context of rendering subset views. One textual view shows the class definition for the graphic_subset_view class, and the other the draw_element method predicate for the graphic_subset class.

A key feature of SPE is its method of 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, it is not possible to automatically infer the correct modification and user assistance is needed. For this reason, updates to views are not immediately performed. Rather, some visual indication of the 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 "*selements*" feature of graphic_subset_view to "*elements*", is reflected in a corresponding textual view by an “update record”, as shown in Fig. 6. This record informs of the change to the base data, and allows the programmer to either make the change, or to select the update record and have SPE make the change to the text (in this case, changing *selements(list(graphic_subset))* to *elements(list(graphic_subset))*).



```
%class_def_start

updates_start(graphic_subset_view).
update(5). % rename feature selements to elements
updates_end.

% graphic_subset_view
%

class(class(graphic_subset_view,
 parents([
  subset_view([
   rename(make_current_view,s_make_current_view),
   rename(process_menu_item,sv_process_menu_item)
 ]),
 attributes([
  selements(list(graphic_subset))
 ]),
 methods([
  draw_subset_elements,
  find_picture,
  add_icon,
  add_blank_icon,
  add_glue,
  create_icon,
  get_icon_details,
  create_glue,
  get_glue_details,
  move_icons,
  make_current_view,
  menus,
  process_menu_item
 ])).

%class_def_end
```

```
%method_start 10

updates_start(graphic_subset_view).
update(2). % add client-server graphic_display_view@visible
updates_end.

/*
 * Draw a graphic subset element.
 *
 */

graphic_subset::draw_element(Element) :-
 Element@view(View),
 View@display(Display),
 Display@lpa_window(Window),
 mv_draw_element(Element,Window).
graphic_subset::draw_element(Element,Window) :-
 mv_draw_element(Element,Window).

mv_draw_element(Element,Window) :-
 Element@get_picture(Window,Picture),
 /* using chg_pic... */
 ( Element@picture(ID) ->
   chg_pic(Window,ID,Picture,(0,0))
 ; gensym(p,ID),
   set_prop(ID,subset_element,Element),
   record_pic(Window,ID,Picture),
   chg_pic(Window,ID,Picture,(0,0)),
   Element@picture:=ID
 ).
mv_draw_element(_,_).

%method_end
```

Figure 6. The Snart Programming Environment.

Another update record is shown in the graphic_subset_Class view. Here, a client-server link has been added between graphic_subset and graphic_display_view indicating that the *draw_element* method of the former makes use of the *visible* feature of the latter. In this case, automatic update of the textual view is not possible as SPE cannot infer the appropriate modification to the *draw_element* method and the user must implement the update.

Going from textual views to graphical is handled in a similar manner. After parsing a textual view, updates to the base 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 work for multiple graphical and textual views of the same information. For example, if the class graphic_subset had a feature moved to its super-class, this would be indicated in any textual views for graphic_subset (both class definition and method predicate views). Any graphical views in which the feature was displayed would be changed by colouring the affected feature connections.

SPE also provides view navigation facilities. These include iconic buttons for quick view selection, view dialogs, and keyword searches for views by name and focus type. Class definitions can be "structure" edited using a dialog-based editing view, and documentation added to classes and features. Graphical views support feature names included with class icons (in a similar manner to the OOATool [Coad 1991]), and provide clusters, groups, and responsibilities for high-level complexity management. Textual views using the MViews editor provide interpretation of update records to modify the class text automatically, hyper-text links to access documentation and other views from the text editor, and the ability to display features inherited by a class together with its own definition.

Programmers typically use graphical views to design their programs, and to visually document a program to enhance readability and browsing. Textual views are used to implement method predicates, Prolog predicates, and to specify additional class definition details, such as renaming of inherited features. Any changes to the program can be made at either the graphical or textual levels, and full consistency between all representations is ensured. After compiling the textual views using the existing Snart compiler, Snart programs can be run and debugged using the Prolog run-time system.

IspelM is a specialisation of MViews, and itself provides a framework for implementing programming environments for object-oriented languages. To specialise IspelM to produce the SPE, we needed to write language-specific parsers and unparsers for textual subset views. The graphical views and base information require little change to support a different language, as common O.O. concepts are captured well at the IspelM level. An interface to the language compiler and run-time system is also necessary for different languages.

6. Summary and current and future work

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 visually programming in Snart.

Other applications of MViews are currently under development. These include:

- A dataflow programming tool after the style of Prograph [Cox 1990]. This will provide an object-oriented dataflow diagramming tool together with a Snart-based interpreter capable of executing the diagrams. The dataflow programs can also be integrated with Snart code allowing a mixture of conventional and dataflow programming.

- A dialog box "painter". This is a visual tool for laying out dialog boxes. The dialog boxes can then be included within a Snart program.

- A visual debugger for Snart programs. This uses a similar approach to the SPE graphical tools, but displays the state of objects rather than classes.

- An entity-relationship diagramming tool. The graphical entities and relationships are translated into relational schema which may be viewed and manipulated in a textual view.

Future applications we envisage for MViews include:

- Specialisations of IspelM for object-oriented languages other than Snart.

- Specialisations of IspelM for object-oriented analysis [Coad 1990; Booch, 1991]. These would provide facilities more abstract than the current design-implementation-maintenance views of IspelM, but should allow progressive refinement through to a full implementation.

- Program visualisation tools to provide a more graphical and dynamic view of program execution than that provided by the visual debugger.

In addition, we expect considerable synergy between the work presented here and another project being undertaken by our group in developing software for the building and construction industry. This latter project aims to develop a common model of a building that various architectural and engineering design tools can interface to throughout the building design/maintenance lifecycle [Amor 1992]. Many of the problems in developing such a model are similar to those faced in developing tools for integrating the various phases of the software design cycle together. In both cases multiple views of the model are essential, and consistency between the views is critical.

Acknowledgements

The financial assistance of Mana Systems Ltd, the Building Research Association of New Zealand, and the University of Auckland Research Committee is gratefully acknowledged.

References

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

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

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

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

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

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

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

Glinert, E.P., and Tanimoto, S.L., 1984: PICT: An interactive, graphical programming environment, *IEEE Computer, 17*, 11, 7-25.

Grundy, J.C., Hosking, J.G., and Hamer, J. 1991: A Visual Programming Environment for Object-Oriented Languages, *Proc TOOLS5,* Prentice-Hall, 129-138.

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

Hosking, J.G., Hamer, J., Mugridge W.B., 1990: Integrating functional and object-oriented programming, *Proc TOOLS3,* TOOLS Pacific, Sydney, pp. 345-355.

Hosking, J.G., Hamer, J. and W.B. Mugridge, 1991: *Kea 1.0 Tutorial Manual*. BRANZ Contract 85-024 Technical Report No18, Department of Computer Science, University of Auckland, 35 p,

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

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

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

Minör, S. 1987: Structured Command Interaction Based on a Grammar Interpreting Synthesizer, *Proc of the Second IFIP Conference on Human-Computer Interaction*, North-Holland.

Moriconi, M. and Hare, D.F. 1986: The PegaSys System: Pictures as Formal Documentaion of Large Programs, *ACM TOPLAS, 8*, 4 pp 524-546.

Myers, B.A., 1990: Taxonomies of Visual Programming and Program Visualization, *Journ. Visual Languages and Computing*, *1,* 1, pp. 97-123.

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

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

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

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

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