In Proceedings of NZCSRSC 1992, Waikato University, 1992

The MViews framework for constructing multi-view editing environments

John C. Grundy and John G. Hosking
Department of Computer Science, University of Auckland, Auckland

ABSTRACT
MViews provides a 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. 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 [1], Prograph [2], and Garden [3]. Useful reviews of visual programming can be found in [4, 5].

In previous work we have developed Ispel, a visual programming environment for object-oriented programming [6]. 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, 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 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. This is followed by an introduction to the MViews architecture and an over-view of the implementation of MViews and IspelM. The paper concludes with a discussion of current and future work.

2. 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 [6], 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 Prolog++ [7]. Fig. 1 shows an example of Snart code.

```
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,
    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.
```

Fig. 1.   An Example Snart class defining Rectangles

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 despatch and attribute access optimised.

3. The Snart Programming Environment

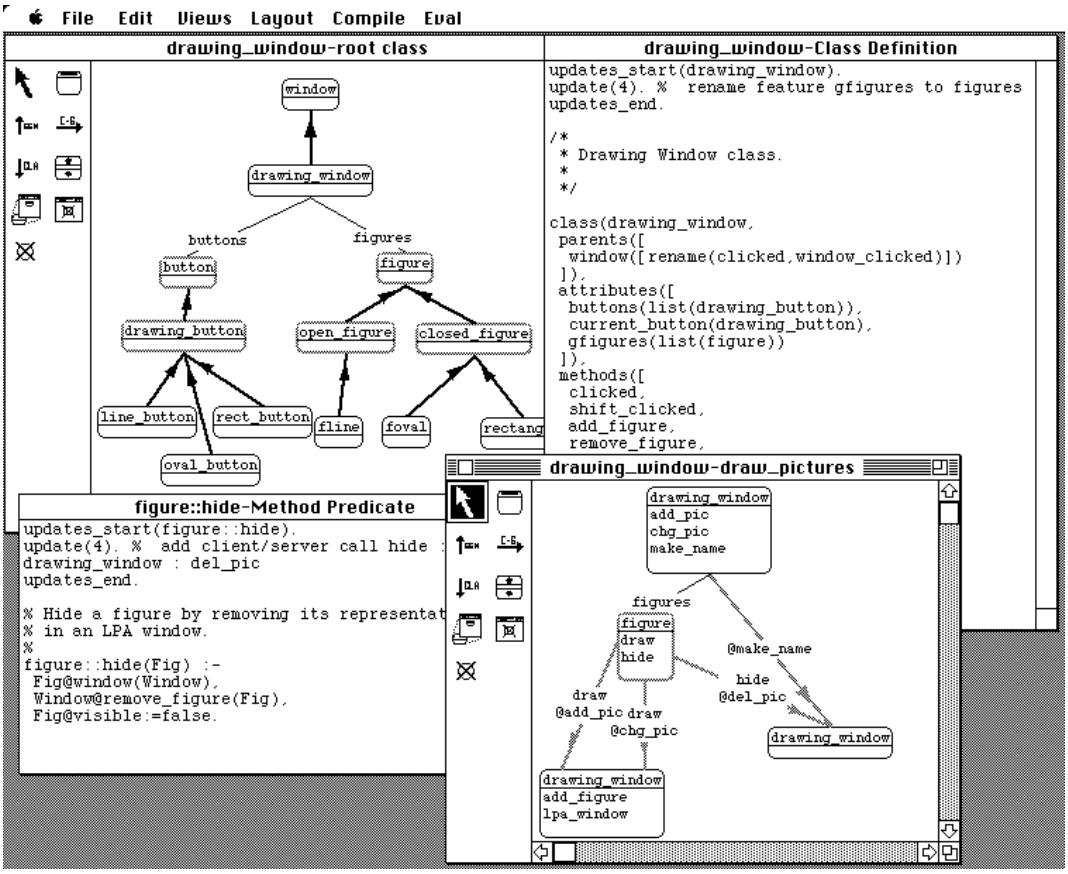
The first application of MViews has been in the development of a visual programming environment for Snart itself. This involved a two step specialisation of MViews. The first step was to generate an Ispel-like object-oriented programming environment in MViews (IspelM). Further specialisation tailored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

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 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 textual and graphical views 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 [8,9].

Fig. 2. 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. A textual view shows the class definition for drawing_window, and the other the hide method predicate for figure.



```
updates_start(drawing_window).
update(4). %  rename feature gfigures to figures
updates_end.

/*
 * Drawing Window class.
 *
 */

class(drawing_window,
 parents([
  window([rename(clicked,window_clicked)])
 ]),
 attributes([
  buttons(list(drawing_button)),
  current_button(drawing_button),
  gfigures(list(figure))
 ]),
 methods([
  clicked,
  shift_clicked,
  add_figure,
  remove_figure,
```

```
updates_start(figure::hide).
update(4). %  add client/server call hide :
drawing_window : del_pic
updates_end.

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

Fig. 2. The Snart Programming Environment.

Update Handling

A key feature of SPE (and MViews) is the 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, such as adding a client-server connection, it is not possible to automatically infer the correct modification 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 Fig. 2. 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))*).

Another update record is shown in the figure Class view. Here, 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. In this case, 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 views 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 are propagated to any combination of multiple graphical and textual views sharing changed information.

View Navigation and Using SPE

SPE provides various view navigation facilities. These include iconic buttons for quick view selection, view dialogs, and keyword searches for views by name and focus type. Textual views using the MViews editor provide hyper-text links and menus to access documentation and other views. Not all views need be cached in memory at once and updates can be propagated to views when they are selected and reloaded. Textual views may contain more than one class definition or method predicate at a programmer's discretion.

Programmers typically use graphical views to design their programs in SPE and to visually document a program to enhance readability and browsing. Textual views are used to implement method predicates, to add 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 parsing textual views the existing Snart compiler is used to generate Snart code and programs can be run and debugged using the Prolog run-time system.

4. MViews and IspelM

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 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 with 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 [6], where multiple views describe overlapping subsets of a base view of an object-oriented program.
    - The dynamic and static views of MELD [10] 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 [11, 12]).
- *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 [13], PECAN [14], Garden [3], 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.

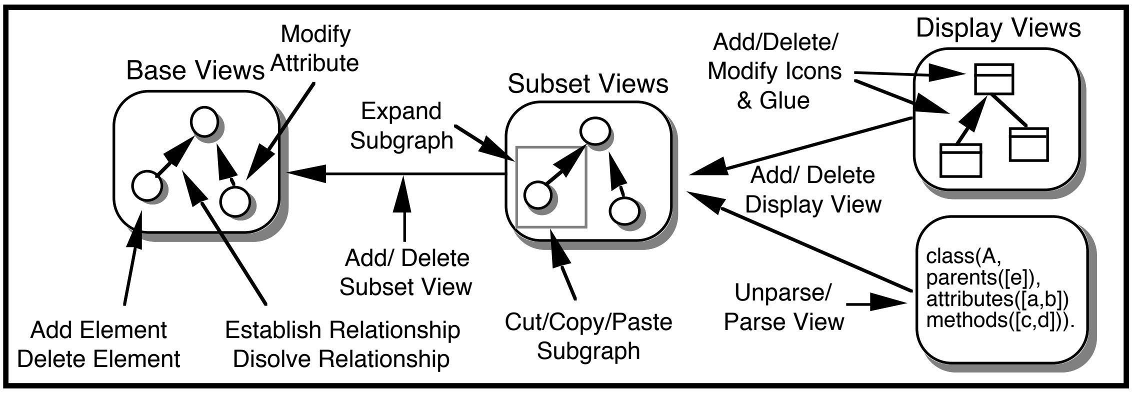
Propagation of change is an essential aspect of MViews’ multiple views. If shared information is modified in one view, a consistency manager propagates the modifications to other views. For example, modification of a display view may alter the base program state. Other views affected by this change must then be updated and redisplayed to provide a consistent presentation of the program across the environment.

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 [15].

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. Fig. 3 shows typical operations affecting 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.



Base Views
Modify Attribute
Expand Subgraph
Subset Views
Add/Delete/ Modify Icons & Glue
Display Views
Add/ Delete Display View
Add/ Delete Subset View
Add Element Delete Element
Establish Relationship Disolve Relationship
Cut/Copy/Paste Subgraph
Unparse/ Parse View
class(A, parents([e]), attributes([a,b]) methods([c,d])).

Fig. 3. Some basic operations on MViews programs.

5. 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 [16] could be constructed, or a specialisable framework implemented as used in Unidraw [17]. 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 [9]. For these reasons the second approach was chosen.

Object-oriented languages foster reuse in various ways [18] 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.

In designing MViews class hierarchies were derived from the formal specification and used to structure the framework. Class responsibilities and services were then determined. 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, generalisations, and client-server relationships as in IspelM.
- Textual and graphical subset views of base data which are partial copies of the base and can be modified by editors or 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.
- 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 structure-editing and text editor facilities. Graphical editors include tools that act upon icons and connector glue to effect changes to subset views. The built-in MViews text editor is used to manipulate textual views but a standard text editor could also be used. The former provides hyper-text links and menus to support view navigation and structured access to subset views.
- Operation recording and histories for subset views that implement undo/redo facilities. These are completely generic requiring little or 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 providing a persistent database storage facility.
- Support for application-specific language semantics and constraint processing.
- Unparsing and parsing support for textual views including parse-tree storage and determination of base view updates via parse-tree changes.

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. MViews and SPE are currently implemented in Snart and run under LPA Prolog on the Macintosh.

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. SPE provides a multiple view programming environment with both textual and graphical manipulation of Snart programs. Design-level and implementation-level changes are kept consistent using the MViews change propagation model.

Other applications of MViews are currently under development. A dataflow programming tool, after the style of Prograph [2], 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. 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 [18] and Kea [19]. Further specialisation of IspelM for object-oriented analysis [20, 21] 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 will provide a more graphical and dynamic view of program execution than that provided by the visual debugger.

Acknowledgements

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

References

[1] Ingalls D, Wallace S, Chow Y Y, Ludolph F, Doyle K, Fabrik: A Visual Programming Environment, in:*Proc OOPSLA ‘88*, (1988) 176-189

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

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

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

[5] Myers B A, Taxonomies of Visual Programming and Program Visualization, in: *Journ Visual Languages and Computing*, **1** (1) (1990) 97-123

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

[7] Pountain R, Adding objects to Prolog, in: *Byte*, August (1990), 64IS-15 - 64IS-20

[8] Ratcliffe M, Wang C, Gautier R J , and Whittle B R, Dora: a structure-oriented environment generator, in: *Software Engineering Journal*, May (1992) 184-190

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

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

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

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

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

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

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

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

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

[18] Meyer B, 1988: *Object-Oriented Software Construction*, Prentice-Hall (1988)

[19] Hosking J G, Hamer J, Mugridge W B, Integrating functional and object-oriented programming, in: *Proc TOOLS3,* TOOLS Pacific, Sydney, (1990) 345-355

[20] Coad P, Yourdon E, *Object-Oriented Analysis*, Second Edition, Yourdon Press (1991)

[21] Booch G, *Object-Oriented Design with Applications* Menlo Park, CA, Benjamin/Cummings (1991)