Constructing Multi-View Editing Environments Using MViews

John C. Grundy and John G. Hosking

Department of Computer Science
University of Auckland, New Zealand
email: john@cs.auckland.ac.nz

MViews abstracts out common features of multi-view editing environments that support integrated textual and graphical programming. It provides a conceptual model and reusable object-oriented framework for constructing such environments. Multiple views of a base document are supported with consistency automatically maintained between each of the views. MViews has been used to construct a visual and textual programming and program visualisation environment for object-oriented systems. Other applications of MViews under development include entity-relationship and dataflow diagrammers with detailed descriptions programmed with text.

1. Introduction

In this paper we describe MViews, a model and framework for supporting the construction of visual programming environments. MViews includes a 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, are constructed by specialising MViews classes.

The advantages of MViews for visual language implementation include its representation of programs and program views as graphs. This allows graph-based languages, typical of visual languages [2], to be represented naturally and graph-based semantics to be represented in the same manner. For multi-view editing, the text/graphics consistency model produces a novel and flexible method for integrating high-level graphical programming and low-level textual programming. The use of a reusable object-oriented framework, rather than an environment generator, produces a very flexible set of building-blocks for constructing environments.

The paper begins with a description of the MViews conceptual model. This is followed by discussion of SPE, a specialisation of MViews, providing a programming environment for an object-oriented Prolog. An overview of the MViews framework and its implementation is presented and the paper concludes with a discussion of current and future work.

2. Multiple view support

Requirements

Visual programming environments that support multiple views require several facilities:

- A graph-based, abstract and flexible program structure and semantics representation [1, 2].
- Textual and graphical view support, as visual programming is typically useful for high-level detail while text is typically useful for low-level detail [9, 11, 13].
- View editing strategies most programmers prefer, such as interactively editing graphics and free-editing text, rather than structure-editing [15].
- An efficient view change propagation mechanism so all views are made consistent after one view is updated. This should support incremental view updates and visual indication of changes that environments cannot automatically carry out [5, 11].
- An undo/redo facility [12] and modification history for program components.
- Program persistency and support for tool integration and extensibility [11].

Existing systems

Smalltalk's Model-View-Controller (MVC) [7] provides a framework where views of a model object are sent "update yourself" messages when the model changes. This often requires a view to redisplay itself whenever its model changes as only an indication of change is propagated [5]. MVC does not directly support incremental view updating nor efficient recalculation of semantic attributes.

Unidraw [14] provides a damage algorithm for incremental view updating and commands implement undo/redo. It does not directly support textual views (except a simple export facility) nor modification histories.

PECAN [12] and GARDEN [13] provide flexible program representations, program persistency and undo/redo. They use an MVC-like update model, however, and do not support modification histories or visual change indication.

Generated environments like Dora [11], Mjølner/ORM [10], and LOGGIE [2] provide structure-oriented editing for views of a program. This editing style has yet to gain wide acceptance among programmers [15].

The `ItemList` [5] provides incremental view updating, similar view and program representation, and undo/redo. It does not provide a natural way of representing visual program structure or semantics, however, nor generic editing mechanisms for view manipulation.

3. MViews architecture

MViews was designed to satisfy the above requirements. A central database holds all information relating to program structure, semantics and different views of a program. Tools communicate via this database and tools for a specific environment, such as text or graphic editors, are either specialised from generic MViews tools, built using the framework, or existing tools reused, such as compilers and run-time systems.

Conceptual foundations

MViews represents programs and views as collections of directed graphs, called *program graphs*. Program graph components are *elements* (graph nodes) or *relationships* between components (labelled graph edges) and both can have named attribute values. As this representation is graph-based, it can represent visual as well as textual languages [1, 2]. A *base view* is used to group program graphs that comprise a shared, canonical representation of a program.

Subset views are program graphs representing a subset of the base program graph and its components. This allows views of a program to be represented and manipulated in the same manner as the program.

Each subset view is rendered either graphically or textually using a *display view*. Display view components render subset view components and include icons, glue and *text forms* (sequential text).

Operations modify a program graph and include component addition and deletion, relationship establishing and dissolving, attribute updating, and view manipulation.

Propagation and recording of change

MViews program graphs are dependency graphs. Every component has zero or more related components that may be affected by a change to itself, called *dependent components*. An operation applied to a component informs these dependents of a change using an *update record*. All operations generate update records, which are, conceptually, lists of values describing the change. For example, an `update_attribute(Name, NewValue)` operation on a component `Comp` will generate the update record `<Comp, update_attribute, Name, OldValue, NewValue>`. Dependent components interpret update records and modify themselves if necessary, possibly generating further update records.

A component may store update records (using a list attribute) to provide a modification history. Update records may also be stored to support undo/redo by sending them back to their generating components for reversal. Update records can also be used to drive data-driven semantics recalculation and support lazy, incremental view updating.

Dependent components know the exact change to their parent, unlike MVC, and thus can make more efficient and precise responses [5]. This also allows more flexible and efficient re-computation after change than data-driven attribute recalculation. Our program graph approach provides a comparable representation to abstract syntax trees but supports graph-based program structures and semantics.

4. The Snart Programming Environment

The first application of MViews has been in the development of the Snart Programming Environment (SPE) for Snart, an object-oriented Prolog [9].

An overview of SPE

SPE supports multiple textual and graphical views of a Snart program sharing common information. This allows the construction of many views, each focussing on different aspects of the program, reducing the cognitive load in understanding a program. Consistency management is employed to maintain data integrity between all views sharing information.

Graphical views are interactively-edited using a palette of tools, menus and dialogues and textual views are free-edited and parsed. This differs from comparable approaches [2, 11] employing restrictive structure-oriented editing [15].

Figure 1 shows SPE editing a simple drawing program. One graphical view shows major inheritance and aggregation hierarchies, the other client-supplier relationships between `drawing_window` and `figure` classes. One textual view shows the class interface for `drawing_window` and the other the `hide` method for `figure`.

Programmers typically use graphical views for analysis and design and for static program visualisation. These views provide class icons with feature names and generalisation, association, and client-supplier relationships. View composition and layout are under the complete control of a programmer.

Textual views are typically used to implement methods and specify additional class interface details. Arbitrary documentation of program components can also be added using textual views. After parsing textual views the Snart compiler is used to generate Snart code.

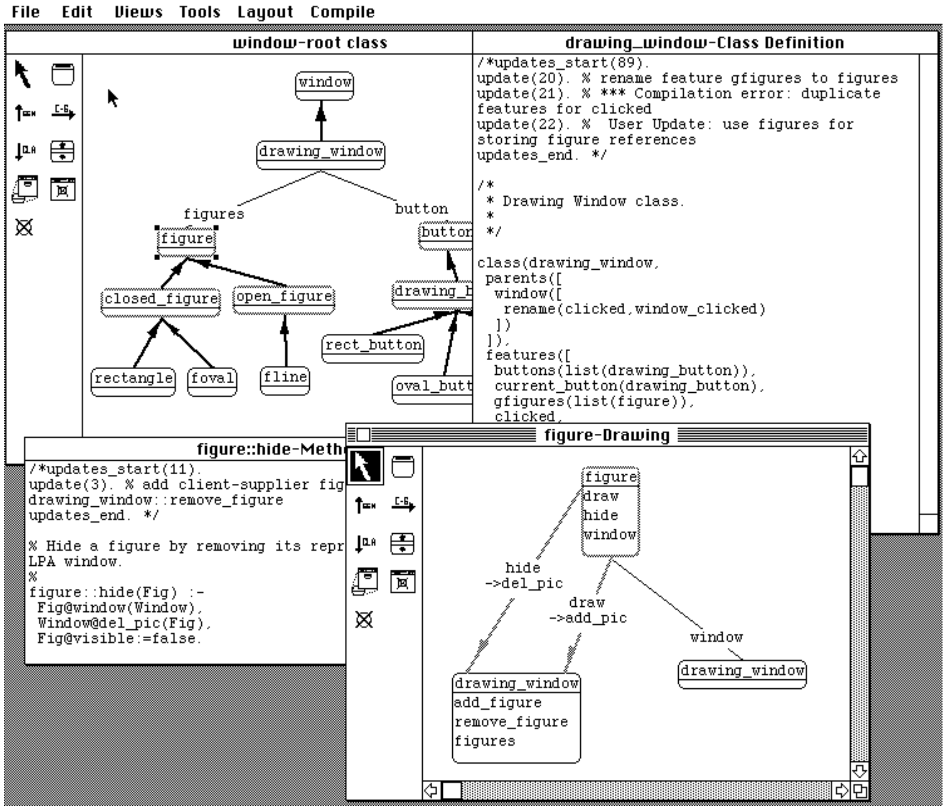

Figure 1. The Snart Programming Environment.

Programmers need to locate information easily from a large number of views and base data. SPE's approach is to use views themselves in a hypertext-like fashion as the basis for browsing, with Prograph-style click-points [4] on icons. Menus are used for textual views for similar view and program browsing facilities.

Programmers can construct additional views for the sole purpose of program browsing, based on information expanded from the base view via dialogues.

Managing change in SPE

MViews provides basic change propagation facilities based on update records which are used by SPE. Some changes to graphical views, such as a feature rename, are applied directly to the view. Other changes, such as deletion of a relationship cause affected components to be highlighted in a different colour (eg red for a deleted feature).

Changes are not immediately made to textual views. Rather, a readable rendering of an update record is inserted into the view and a programmer can accept, implement, or reject the update.

For example, from Figure 1 the first update record in the `drawing_window` view indicates that the `gfigures` feature has been renamed to `figures` in another view. The programmer can:

- accept the change and have SPE modify the text (changing `gfigures` to `figures`)
- implement the change manually
- reject the change, causing it to be undone

In some cases, such as the addition of client-supplier glue in a graphical view, it is not possible to automatically update a textual view. Such an example is shown in the `figure::hide` view of Figure 1. For this change SPE cannot infer the appropriate modification to `hide` and the programmer must change the method text.

Update records may also be used to inform users of errors and to document changes via “user defined” updates. A compilation error and user-defined update are shown in the `drawing_window` class definition in Figure 1.

One important consequence of the update record approach to consistency management is that the collection of update records for a component provides a modification history which SPE permits to be browsed and modified.

Runtime support

Programs can be run and debugged from within SPE. A rudimentary visual debugger allows object data to be displayed and navigated between using graphical object views.

MViews can also be used to produce animations of executing Snart programs. Snart provides a dynamic object tracing mechanism where individual objects and some of an object's features can be spied. MViews uses this to produce update records equating to object method calls and attribute assignment which can be used to drive an animation.

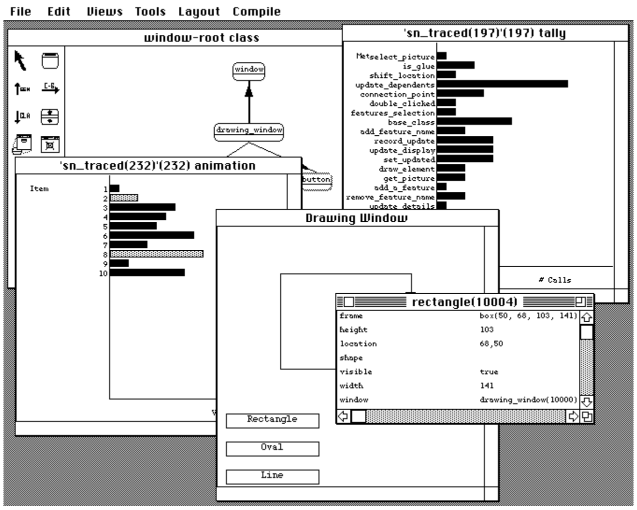

Fig. 2. Program visualisation in SPE.

For example, Figure 2 shows an animation of a sorting algorithm, a tally graph showing method calls to an object, and an object data view. Sort animation detects compare/swap method calls on a sorting object. The tally graph is a subset view of a hashtable which counts another object's message receipts. Both use the same display view (in this case a bar graph). We are extending SPE's visualisation capabilities to include graphical inter-object references, similar to [6], and to support control flow visualisation.

5. Design and implementation

To produce a reusable implementation for MViews a programming environment generator with its own specification language [2, 10] or a specialisable object-oriented framework [14] could be implemented. Many aspects of an interactive environment, such as good editor interfaces, require specialisation and fine-tuning on difficult to provide with a generator. For these reasons a framework approach was chosen.

The MViews architecture

A collection of abstract classes provide a framework for MViews-based environments. Different kinds of MViews components are modelled as a class hierarchy, operations are implemented as method calls and attributes and relationships as objects.

Base program component classes hold program structure and semantics data and relationships connect these components. Basic component operation methods can be augmented with more complex operations in sub-classes Subclassing can also over-ride default method implementations to provide application-specific language semantics and constraints. Update records are propagated and stored as objects with attribute values describing the update.

Subset component classes support the same operations and update propagation mechanism as base components with extra methods for view support. Subset views record subset component updates for undo/redo and interface to display views and external systems.

Textual and graphical display views group renderings of subset view components as icons, glue or text forms. Display views and components are defined as specialisations of subset views and components.

Graphical views use interactive editing of graphical icons and glue. The built-in MViews text editor is used to free-edit textual views. Textual views are parsed, using application- specific parsers, to update MViews base components.

A Snart implementation of MViews

MViews has been implemented in Snart and this provides a reusable framework of Snart classes. Components and relationships are implemented as Snart classes, attributes stored as Snart object attributes, and operations implemented by Snart methods.

Systems such as SPE are constructed by appropriately specialising these framework classes. For example, a `base_class` is specialised from `base_component`, a `class_icon` from `graphic_icon`, and a `class_code_view` from `textual_view`. Environments define their own attributes, relationships and operation methods but reuse MViews' multiple views, consistency management and persistency systems.

The IspelM framework and SPE

Development of SPE was a two step process. Firstly IspelM, a framework which supports programming environments for object-oriented languages (i.e. is language independent and reusable for different languages), was specialised from MViews. Further specialisation tailored IspelM for programming in Snart producing SPE. IspelM defines classes specialised from MViews which implement:

- Object-oriented program representation including class and feature components, and generalisation and client-supplier relationships.
- Graphical display views and components for describing and manipulating class relationships with most methods for interactive editing inherited from MViews classes.
- Textual display views and text form classes with text form management, editing, and parsing and unparsing primitives inherited from MViews classes.

SPE specialises IspelM to support development of Snart programs. SPE classes define Snart-specific parsers and unparsers for textual display views, saving and loading support for Prolog code, and an interface to the existing Snart compiler and run-time system.

Experience with MViews systems

The development of SPE and IspelM was considerably easier when compared with Ispel, which was implemented without framework support [8]. A program visualisation system, an entity-relationship modeller and a dataflow diagrammer are currently under development using MViews. These support multiple textual and graphical views of quite diverse visual programming languages and related textual information. Preliminary results suggest development of these systems is greatly simplified by using the MViews model and framework.

Systems which support multi-view editing allow programmers to describe information using the most convenient representation and level of abstraction [3]. Use of multiple textual and graphical views in SPE demonstrates the need for visual indication of view changes and the ability to automatically apply these changes. Together with component change histories, this provides a novel solution to integrating low-level textual programming with high-level visual programming and graphical program visualisation. Use of SPE indicates the MViews approach shows great promise for integrating these textual and graphical paradigms [9].

6. Summary and current and future research

We have described MViews which supports: program structure and semantics representation as program graphs; multiple textual and graphical views; consistency management via update record propagation; update record storage for undo/redo and modification histories; generic routines for component persistency; and a consistent user interface with external tool interfacing. MViews has been reused in the development of IspelM, an environment for object-oriented programming. SPE, a specialisation of IspelM, supports textual and graphical manipulation of Snart programs. MViews provides abstractions and a framework so new environments can support multi-view editing by simply reusing MViews.

Other applications of MViews currently under development include: visual debugging using diagrams to illustrate object references; a dataflow programming tool, similar to Prograph [4], which can be integrated with textual views allowing a mixture of textual and dataflow programming in SPE; and an entity-relationship diagramming tool with textual relational schema.

Future applications for MViews include:

- Specialisations of IspelM for object-oriented languages other than Snart
- Better IspelM support for analysis and design.
- Recording of update records in groups with arbitrary undo/redo to support flexible version control. With configuration management this could be used to support distributed multi-user software development.

Acknowledgments

The financial assistance of the University of Auckland Research Committee is gratefully acknowledged. John Grundy is supported by IBM, William Georgetti and New Zealand Universities Postgraduate Scholarships.

References

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

[2] Backlund, B., Hagsand, O., Pehrson, B. (1990): Generation of Visual Language-oriented Design Environments, In *Journal of Visual Languages and Computing* **1** (4), 1990, 33-354.

[3] Brown, M.H. (1991): Zeus: A System for Algorithm Animation and Multi-View Editing, In *Proc of IEEE Symposium on Visual Languages*, 1991, 4-9.

[4] 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.

[5] 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.

[6] Fenwick, S., and Hosking, J.G. (1993): *Visual Debugging of Object-Oriented Systems*, Departmental Report #65, Computer Science Department, University of Auckland.

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

[8] 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.

[9] Grundy, J.C., and Hosking, J.G. (1992): MViews: A Framework for Developing Visual Programming Environments, In *Proc TOOLS Pacific '93*, Prentice-Hall.

[10] Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minor, S., Oscarsson, D., Taube, M. 1990: An Overview of the Mjølner/ORM Environment, In *Proc TOOLS '90*, Prentice-Hall.

[11] Ratcliffe, 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.

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

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

[14] Vlissides, J.M., Linton M.A. (1989): Unidraw: A framework for building domain-specific editors. In *Proc ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989, 158-167.

[15] Welsh, J., Broom, B., Kiong, D. (1991): A Design Rationale for a Language-based Editor, Software - Practice and Experience, **21** (9), 1991, 923-948.