Multiple textual and graphical views for Interactive Software Development Environments

John Collis Grundy

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

University of Auckland   June 1993

Abstract

Diagram construction can be used to visually analyse and design a complex software system using natural, graphical representations describing high-level structure and semantics. Textual programming can specify detailed documentation and functionality not well expressed at a visual level. Integrating multiple textual and graphical views of software development allows programmers to utilise both representations as appropriate. Consistency management between these views must be automatically maintained by the development environment.

MViews, a model for such software development environments, has been developed. MViews supports integrated textual and graphical views of software development with consistency management. MViews provides flexible program and view representation using a novel object dependency graph approach. Multiple views of a program may contain common information and are stored as graphs with textual or graphical renderings and editing. Change propagation between program components and views is supported using a novel update record mechanism. Different editing tools are integrated as views of a common program repository and new program representations and editors can be integrated without affecting existing views.

A specification language for program and view state and manipulation semantics, and a visual specification language for view appearance and editing semantics, have been developed. An object-oriented architecture based on MViews abstractions allows environment specifications to be translated into a design for implementing environments. Environment designs are implemented by specialising a framework of object-oriented language classes based on the MViews architecture. A new language is described which provides object-oriented extensions to Prolog. An integrated software development environment for this language is discussed and the specification, design and implementation of this environment using MViews are described. MViews has also been reused to produce a graphical entity-relationship/textual relational database schema modeller, a dialogue painter with a graphical editing view and textual constraints view, and various program visualisation systems.

Acknowledgments

My supervisor John Hosking has been a tremendous guiding influence during the research for and writing of this thesis. He has helped pick my spirits up when down, encouraged me when things have gone well and kept a sometimes way-ward spirit on the right path too many times to mention. I have greatly valued his friendship, constructive criticism and excellent supervision during the course of this work. I also look forward to continuing professional and personal relationships for many years to come. Thanks for everything, John.

I would also like to thank various colleagues for their helpful comments about my research. Particular thanks go to Robert Amor, Stephen Fenwick, Rick Mugridge and John Hamer from the University of Auckland and James Noble from Victoria University of Wellington.

The staff and students of the Department of Computer Science at Auckland University have made my time as a post-graduate student both enjoyable and personally and professionally enriching.

I would like to thank my friends and family for all their support over the years. I particularly would like to thank my parents, Irene and Ray, for giving me so much love and support all my life and for the educational opportunities they never had. Thanks also go to my parents-in-law, Jannie and Mike Visser, my sisters and brother Heather, Jenny and Mike, and my sister-in-law Alice, for all their love and support.

The financial assistance of the Vice-Chancellors' Committee, IBM New Zealand Ltd, and the William Georgetti Scholarship Committee are gratefully acknowledged. I would also like to thank Professor Bob Doran for helping to ensure my financial well-being with work for the Department of Computer Science over the past three years.

The biggest thank you goes to my wife, Judy, for the incredible amount of love, caring, kindness and assistance she has given me. I can not begin to express the love and appreciation I have for her for being so supportive during what has been some very good and some quite trying times. I would also like to thank my daughter, Stephanie, for blessing Judy and I with such a precious gift as herself. This thesis is for you, Judy and Stephanie. Thank you so very much.

To Judy and Stephanie:

your love and support have made this work possible.

Contents

Contents

Contents   Page xiii

Chapter 1

Introduction

This chapter discusses the main rationale for this research, the integration of textual and graphical views of software development. It also discusses the requirements of integrated software development environments (ISDEs) and the importance of a common set building blocks for these systems. The major goals of this research are outlined and include production of a reusable model for ISDEs, development of an ISDE for an object-oriented language by reusing this model, and reuse of the model for other systems to demonstrate its flexibility. The major research contributions of this thesis are summarised to illustrate how these goals have been fulfilled.

1.1. Rationale for Research

Programming environments (PEs) assist programmers to implement and debug programs by providing tools which make the task of program construction easier (Dart et al 87). Software development environments (SDEs) subsume programming environments and provide tools for various software management tasks such as analysis, design, implementation, debugging, maintenance and version control (Dart et al 87, Meyers 91). Software development environments can use graphical program representations for analysis, design, visual programming, static and dynamic program visualisation and debugging, documentation, and maintenance. Textual program representations can be used for program implementation and documentation. An ISDE must provide automatic consistency management between different program representations that share information. It should also allow new or existing tools to be integrated into the environment (Meyers 91).

1.1.1. Textual vs. Graphical Program Construction and Representation

This thesis was primarily motivated by a desire to integrate multiple textual and graphical views of programs. Graphical views of program structure and semantics are used by many analysis and design methodologies (Fichman and Kemerer 92, Henderson-Sellers and Edwards 90, Shlaer and Mellor 88) and CASE tools provide editors for constructing such diagrams (Coad and Yourdon 91, Wasserman and Pircher 87, StructSoft 92). These views of program component relationships allow designers and programmers to reason

about large software systems at a high level of abstraction. Graphical program component representations are also used to support program browsers (O'Brien et al 87, Fischer 87, Symantec 90) and static and dynamic program visualisation and documentation (Myers 90, Kleyn and Gingrich 88, Wilson 90). Visual programming languages use diagrams to specify the structure and execution semantics for programs (Myers 90, Ambler and Burnett 89, Raeder 85).

In contrast, textual program representations are usually used to specify low-level detail about program structure and semantics. Most traditional programming languages are text-based and text is used to specify data structures and functionality. Text can be used to specify some higher-level aspects, such as class contracts in Eiffel (Meyer 88 and 92). Text is also used to document software systems, in terms of both programmer and end-user documentation.

Some programmers find linear sequences of textual characters do not convey the high-level structure of programs as well as equivalent graphical representations (Ambler and Burnett 89). As high-level program component semantic relationships are almost always graph-based, a graphical representation often expresses these relationships well (Reiss 90b, Myers 90). Conversely, some programmers find graphical program representations have poor power of expression for some sequential control structures and expressions (Myers 90, Reiss 90b, Vlissides 90). Thus an environment should ideally allow programmers to choose the form of program representation and editing they find most appropriate for a particular task.

1.1.2. Integrated Textual and Graphical Programming

As graphical and textual programming styles suit different programmers' requirements for program construction, a natural approach might be to support both representations within a programming environment. A programming environment could have multiple views of a program, some being graphical, diagrammatic representations and others being textual representations. Programmers could then select the representation desired for design and implementation tasks.

Many programming environments support multiple textual and/or graphical views of parts of programs (Reiss 85, Reiss 87, Kaiser and Garlan 87, Backlund et al 90, Ratcliffe et al 93). Systems providing integrated graphical and textual views, such as Dora (Ratcliffe et al 92) and PECAN (Reiss 85), typically use structure-oriented editing of views. Program components affected by editing operations propagate changes to all affected views to keep all views consistent. Structure-oriented editing, however, tends to be a rather restrictive approach to editing programs and is not very suitable for editing low-level expressions and control-statements (Welsh et al 91), nor for graphical diagram construction (Arefi et al 90). Programmers generally find structure-oriented editing difficult and unnatural for

low-level detail and such editors, while well researched in recent years, have yet to gain a wide-spread acceptance (Welsh et al 91, Whittle et al 92, Minör 90, Arefi et al 90).

1.1.3. Integrated Software Development

Software development environments use graphical and textual representations of programs for more than just design and implementation. An integrated software development environment supporting multiple textual and graphical views should be able to reuse views for different tasks (such as class diagrams for object-oriented analysis and design (Coad and Yourdon 91, Fichman and Kemerer 92) and class textual descriptions for programming and documentation (Meyer 88)). New program representations and their editing tools should be integrated (both data representation and user interface) without affecting existing view and tool data storage and behaviour (Meyers 91, Reiss 90a, Wang et al 92).

Different integrated software development environments support these common facilities. This suggests a need for a reusable model for ISDEs. ISDE production is time-consuming and difficult and, given their common facilities, a reusable model would be of great benefit (Backlund et al 90, Meyers 91, Wang et al 92).

1.2. Goals of Research

The initial goal of this research was to produce an ISDE supporting integrated textual and graphical views of a program with consistency management. This environment would be unique, however, in that it would utilise free-edited textual views and interactively edited graphical views of programs. These editing styles are the ones most often used by programmers for each kind of representation. An object-oriented language would be the target language for the environment as object-oriented language structure is very suitable for visual programming and representation (Myers 90, Ratcliffe et al 92).

A second major goal was to develop a set of reusable building blocks to assist in the construction of integrated software development environments. This model should support flexible program representation (including support for representing graph-based, visual languages) (Arefi et al 90, Backlund et al 90), should provide both language-specific structure and semantics support (Reps and Teitelbaum 87), and support integrated, multiple and textual views of software development (Meyers 91). The model should also have abstract support for program data persistency (Minör 90, Wang et al 92) and tool integration and extensibility (Meyers 91, Reiss 90a).

Using this model as a basis, ISDEs should be abstractly specified and an aim was to produce a specification language suitable for this task. This language should be able to be formally reasoned with to ensure an environment specification correctly uses the

abstractions defined by the model. To implement an environment, a formal specification could be used to generate an implementation (Reps and Teitelbaum 87, Backlund et al 90, Magnusson et al 90) or a reusable collection of classes or abstractions specialised or used (Linton et al 88, Vlissides 90, Haarslev and Möller 90). To demonstrate the ISDE model developed is realisable, a goal was to implement the representative ISDE described above using one of these two approaches. To illustrate that this model is reusable for other environments and applications, the final goal was to produce one or more other systems by reusing the model and its implementation.

1.3. Contributions of Research

This thesis provides a range of contributions to the field of software development environment research. These contributions include:

- MViews, a model for ISDEs, has been developed. MViews represents programs and partial views of programs as graphs and program structure and semantics are specified and represented in the same graph-based form. Views are rendered and manipulated graphically or textually. Change propagation between program and view components is supported by a novel update record propagation mechanism. This propagates a record of the exact change to a component to other components dependent on its state. Tools are interfaced to a canonical representation of the program using views.

- Two specification languages for MViews-based environments allow such environments to be defined in an abstract style. A textual specification language defines the state of programs and views of a program. It also defines how program components may be changed using a basic set of graph manipulation operations and how components respond to changes to other components. A visual specification language defines the appearance of program views, view components and dialogues. Editing operations on these visual entities are propagated to the program state defined by the textual specification language and vice-versa. This defines the interaction between program state change, view component appearance and view component editing.

- An object-oriented architecture is used to design an implementation for MViews-based environments. Currently, environment implementers translate environment specifications into this architecture by hand. A framework of classes has been developed which implements this architecture. New environments are implemented by specialising these framework classes based on the environment's object-oriented design.

- A simple language has been developed which provides object-oriented extensions to Prolog. An ISDE for this language has been produced which reuses MViews to support its multiple textual and graphical views. Graphical class diagram views are used for analysis, design, browsing and static

visualisation of programs, while textual class code views are used for class interface specification, method implementation and detailed documentation. Class diagram construction supports visual structure programming while textual views are free-edited and parsed to modify program detail. Full consistency management between views is supported using MViews' novel update record and object dependency mechanisms.

- Other systems have been developed that reuse MViews. An entity-relationship modeller provides graphical entity-relationship diagrams and textual relational database schemas. These are kept consistent using MViews' consistency model. A dialogue painter provides a graphical dialogue painter with textual dialogue constraints and semantics processing views. Changing one dialogue representation propagates changes to other views allowing a dialogue's appearance to be specified graphically while constraints and default values are specified using text. Program visualisation systems provide a visual debugger, method call tally graph view, and sorting algorithm animation view.

1.4. Thesis Organisation

The following chapters are organised thus:

- Chapter 3 defines a simple language, Snart, an object-oriented extension to Prolog. Snart provides an example language to construct an ISDE for and provides a language to implement this environment. Object-oriented languages are focused on in this research as they are particularly appropriate for graphical representation (Wilson 90).

- Chapter 4 describes a programmer's perspective of the Snart Programming Environment (SPE). SPE is a representative ISDE for analysing, designing, implementing and maintaining Snart software. SPE provides graphical, interactively edited views for analysis, design, static program visualisation and program browsing. Textual views are free-edited and parsed, and support detailed software documentation and program implementation. Full consistency management between different views is supported using a novel update record mechanism. These update records also provide a change documentation facility for program components. SPE is described to illustrate the kinds of views and facilities useful for ISDEs. It is used in the following chapters to illustrate the kind of environments this research aims to facilitate modelling and construction of.

- Chapter 5 discusses some requirements for systems used to construct ISDEs and defines the MViews model for ISDEs. MViews introduces a novel technique for representing program structure and semantics and views of a program. A novel update record mechanism for propagating and documenting change in an ISDE is also introduced. MViews provides a model for specifying ISDEs and supports

graphical and textual views of information. Tools are interfaced to a common program data repository and tools share a common set of user interface abstractions. To specify environments that use the MViews model two specification languages are developed. A textual language specifies environment program and view state and the modification semantics of this state. A visual language uses examples and simple visual "programming" to specify editing tool appearance and functionality.

- Chapter 6 describes an object-oriented architecture for designing new environments based on the model of Chapter 5. The developer of an environment translates a formal specification for the environment into a design which reuses this architecture. The architecture provides a class hierarchy with classes providing abstractions based on the MViews model of environments. This architecture allows an environment implementer to design a new environment in a manner consistent with the MViews model.

- Chapter 7 provides a framework of Snart classes used to implement environments modelled and designed using MViews and its architecture. To implement an environment, Snart classes are defined which inherit much of their data and behaviour from the Snart framework. This illustrates that new environments based on the MViews model can be practically realised.

- Chapter 8 shows how SPE can be designed using the architecture of Chapter 6 and implemented using the framework of Chapter 7. SPE itself can be generalised to IspelM, a generic ISDE for object-oriented languages. This chapter demonstrates the advantage of a model, architecture and framework which supply most of the data modelling and functionality for developing ISDEs supporting multiple textual and graphical views of software development.

- Chapter 9 illustrates that MViews can be used to develop a diverse range of environments and systems. Some of the systems developed include an entity-relationship diagrammer with textual relational database schema, a dialogue painter with textual constraint specification, program visualisation views and visual debugging views. These and other systems illustrate the flexibility of MViews and the usefulness of its architecture and framework.

- Chapter 10 summarises the contributions of this research and draws conclusions about the usefulness of MViews and its derivatives. Aspects of ISDEs which are not well supported by MViews are identified and future research proposed to satisfy these requirements.

Chapter 2

Related Research

This chapter reviews current research on programming environments (PEs) and integrated software development environments (ISDEs). We begin by using Unix-style textual programming environments as a base example of PEs. Such environments are generally not well integrated, not very interactive, and generally text-based. Various improvements to text-based environments have been made. These include systems which generate integrated, interactive environments from formal grammars, support integration and extension of text-based programming tools, and provide distributed, multi-user, text-based software development environments.

Graphical (or "visual") programming environments, CASE tools, and program visualisation systems use graphical pictures, rather than textual character sequences, to describe programs. CASE tools provide drawing-style editors for constructing diagrams for analysis, design and documentation of software. Visual programming environments use diagrams to specify the data and functionality of programs and these diagrams can be executed to run a program. Program visualisation systems display program data and functionality at different levels of abstraction. These include visual debugging systems, static and dynamic program structure and functionality visualisation systems, and algorithm animation systems.

Some recent efforts at producing ISDEs have attempted to combine textual and graphical modes of software specification and programming. Another general trend for producing ISDEs for large-scale software production has been to produce environments which provide well-integrated tools (at both the data and user interface levels) with provision for environment extensibility (changing tools or integrating new tools into the environment). These are usually conflicting aims as good extensibility often makes good integration difficult and vice-versa. Different researchers have tended to stress one goal over the other, depending on their view of which is the most important. The final sections of this chapter briefly outline several important requirements for ISDEs this thesis addresses. An overview of the thesis structure is given to illustrate how these requirements are met.

2.1. Textual Programming

2.1.1. Unix-style Environments

Traditional programming environments are based on an edit-compile-run sequence of program development (Dart et al 87). A good example is the traditional approach to C programming using Unix systems. A command shell, such as the c-shell (`csh`) is provided by the operating system (Unix) and various programming "tools" are directly invoked using a command-line interpreter provided by the shell (Reiss 90a). A tool is typically some environment function or program used to perform a specific programming or software development task (Dart et al 87). Such tools might be a standard text editor (such as `vi` or `emacs`) and a standard C compiler (such as `cc`). Programmers edit their programs, compile their programs, then execute their programs (which are compiled to an executable invoked from the command line). Additional programming tools, such as a version control system (for example, RCS or SCCS), a help system (such as `man`), and a debugger (such as `dbx`), can be supplied as executable programs or groups of executable programs.

The main problem with such environments is that they are poorly integrated at both the data and user interface levels. All data is typically stored in Unix files and different tools may have no access to data used by other tools. Tools may even duplicate data and store it in different, incompatible formats. This can easily lead to incompatible program representations and lack of data portability between tools. Even if graphical browsing tools are supported, programming is almost exclusively text-based with little or no ability to visualise complex program structures in diagrammatic forms.

Unix-style environments might be viewed as the most extensible kind of environment. New tools can be added at will but tool integration may only be via the command line interpreter (with possibly no, or very rudimentary, data transfer between tools). Even when graphical user interfaces are supported by the operating system, these are usually restricted to pull-down menus and windows which contain textual command shells and editing windows. Some environments, such as Eiffel (Interactive 89), provide tools which make use of graphical user interfaces to support program browsing. Generally these simply provide a slightly higher-level access to the Unix command-line interpreter with tools invoked with menu commands rather than command-line instructions.

Program development in such environments tends to be rather batch-style with a program edited, compiled to detect syntax and semantic errors, and then debugged and re-edited. There is generally no direct support for software analysis or design, and maintenance uses the same text-based edit-compile-run cycle. Programs are usually re-executed after an edit and thus testing can be very time-consuming. Environment tools are usually implemented from scratch (using C or command-line scripts) and thus these environments require a great effort to develop.

2.1.2. Purpose-built, Tightly Integrated Environments

An improvement over Unix-style environments are purpose-built, tightly-integrated environments (Dart et al 87, Reiss 90a). These typically have the editor, compiler and run-time system integrated with a graphical user interface and common data storage. A typical example is THINK C on the Macintosh (Symantec 89). C programs are edited using one or more Macintosh window-based text editors and then compiled using a menu option. Programs are run in the same environment and a source-level debugger allows C code to be viewed as program statements are debugged. Turn-around time between debugging, compiling and editing tends to be much quicker as the "tools" are always in memory and executing, and tools share a common user interface behaviour.

While nice to use, from a programmer's point of view, these environments have major disadvantages. The programming effort required to produce them is enormous (Reiss 90a, Dart et al 87) and while they are highly integrated, they usually support very limited (or no) extensibility. The "tools" comprising the environment are often only one tool (the whole environment) which supports every programming task from editing to debugging. Thus the program which implements the environment must be changed to support new or different tool functions, often a very difficult thing to do. While the environment as a whole has good user interface and data integration, other environments and tools can not usually access this data directly, nor be invoked by or invoke the environment's user interface.

2.1.3. Tightly Integrated, Extensible Environments

Some languages, such as LISP, Smalltalk and Prolog, have language interpreters and environments which include either a command-line interpreter for language constructs, or are written in the target language itself. Some text editors support editor extensibility by providing a language which can be used to extend the editor's functionality (for example, the Unix `emacs` editor which has a LISP interpreter). To extend the environment's functionality, a new "tool" can be implemented using the target programming language and then invoked by the environment in the same manner as other programs. Graphical user interfaces for these environments, such as those provided by LPA MacProlog (LPA 92), Smalltalk (Goldberg 84), and InterLISP (Kleyn and Gingrich 88), provide programmers with the same high-level tool interfaces as purpose-built environments.

While these environments are extensible and integrated, they usually have the same problem of data and user interface integration within the language interpreter itself. Thus existing tools not implemented in the target language supported by the environment are difficult to integrate (but still somewhat easier to integrate than with purpose-built environments) (Reiss 90a). While environments like LPA MacProlog and Smalltalk provide good programming facilities, such as cross-referencing information and multi-

window editing, they do not generally provide software development tools for analysis and design. If such tools are supported, they are usually rather limited browsing tools or very simple "template generators". An example is the LPA MacProlog MacObject editor which generates code for Prolog++, an object-oriented extension to LPA MacProlog. This only supports simple object inheritance diagrams and its diagrams are not automatically updated when the Prolog++ code it represents is changed.

2.1.4. Generated Environments

As the effort of producing a programming environment is a large task, many researchers have attempted to provide declarative specification languages for languages and their environments. These have usually been based on the abstract syntax of a language, as opposed to traditional batch-style compilers, like the Unix C and THINK C compilers, which use the concrete syntax of a language. Programs are typically "synthesised" using language construct "templates" and "structure-oriented editing". An early example is the Cornell Program Synthesizer (CPS) (Reps and Teitelbaum 87). Program control statements and data declarations are defined by successively expanding and filling in templates based on an abstract syntax definition of Pascal. In addition, the static semantics for a program under construction are checked by an incremental attribute grammar specified around the abstract syntax (Reps and Teitelbaum 87). Expressions are free-edited and then parsed rather than structure-edited.

An abstraction of the CPS is the Synthesizer Generator (Reps and Teitelbaum 84). This allows environments to be generated from operator-phylum abstract syntax grammars and attribute grammars based around these abstract syntax specifications. Editing is via structure-editor template commands (for all parts of a program) which ensures a syntactically incorrect program can not be derived. Mercury (Kaiser et al 87) uses the Synthesizer Generator to provide distributed, multi-user programming environments that support automatic propagation of module interface changes among several users. Neither of these programming environment generators support any other software development tasks than textual, structure-oriented editing with incremental static semantics checking.

These systems allow language structure and static semantics to be specified very abstractly and in a declarative manner. New environments can be quickly defined and generated based on a common set of user interface, data storage, and data recomputation abstractions. Environments produced in this way are not very extensible, however, and use a restrictive editing style.

The UQ2 editor (Welsh et al 91) attempts to overcome some of these problems by allowing users to determine the kind of editing for a given program construct. New editors are specified and generated from grammars but provide various extensions to support more conventional free-editing styles as well as structure-oriented editing and incremental

parsing. These editors do not directly support integration with other tools but do support flexible program documentation and textual browsing capabilities.

2.1.5. Text-based Software Development Environments

Some generated environment efforts provide support for tool integration and extensibility. An early effort was the Gandalf project (Notkin 85). Gandalf supports structure-oriented editing using the ALOE editor but also has tools for project management, version control, and other software development activities. All these tools are text-based, however, although derivatives of the Gandalf Project, such as the GNOME Project, use diagrams to illustrate parts of software systems (Myers et al 88).

Centaur is a generic software development environment which generates environments from formal specifications (Borras et al 88). Centaur uses textual editors based on abstract syntax grammars and automatically translates these trees between persistent and in-core forms as necessary. Centaur also has semantics and concrete syntax specification languages and a graphical user interface for editing tools. Tools for project management and documentation can be defined as well as program editors.

MELD (Kaiser and Garlan 87) is a declarative language for specifying tool interfaces ("static" views), views of a program ("dynamic" views), structure-oriented text editors, and static and dynamic language semantics. The main advantage of MELD is its declarative specification (based on language abstract syntax and action equations (Kaiser 85)) from which tool data storage, tool interfaces, and tool editors are generated. Static tool views automatically translate operations on one tool interface to equivalent operations on other views of this tool interface. Dynamic views allow partial views of data to be specified using a database-like query and these views are automatically updated as the data they model changes. MELD does not provide any direct support for graphical diagramming tools.

Mjølner/ORM environments (Magnusson et al 90) use interpreted abstract syntax grammars to generate textual structure-oriented editors for programming languages. Mjølner/ORM also provides an object-oriented attribute grammar language for specifying static and dynamic language semantics in a very abstract form. Mjølner/ORM environments provide improved editing facilities and representations over the Synthesizer Generator (Minör 90, Whittle et al 92) but have no multiple-view or graphical view support. Some graphical tools can be used, such as a pictorial representation of software version control, but these are currently hand-coded and then interfaced to the environment. Mjølner/ORM environments support tool extensibility via a "back-bone" (Magnusson et al 90, Minör 90) which supports dynamic loading of tools and tool data, data storage in abstract syntax forms using Unix files, and data integration of new tools which use Simula objects.

2.2. Graphical Programming and Visualisation

2.2.1. CASE Tools

Most software analysis and design methodologies use diagrams to model the high-level analysis and detailed design of software systems (Fichman and Kemerer 92). Such methodologies include Yourdon Structured Analysis (Yourdon 89), Shlaer and Mellor Object-Oriented Analysis (Shlaer and Mellor 88), Object-Oriented Structured Design (Wasserman et al 90), and Entity-Relationship Modelling (Chen 76). Software developers can use multiple diagrams to show different views of software at different levels of abstraction. These diagrams generally illustrate the structural and semantic relationships between different high-level aspects of a software system better than textual representations.

CASE tools provide graphical editors supporting the construction of these analysis and design diagrams (Chikofsky and Rubenstein 88). They usually provide consistency management between different views to ensure a software developer has a consistent view of the software system under construction. Software thru Pictures (Wasserman and Pircher 87) provides various views which support dataflow analysis, structured analysis and detailed object-oriented design. The OOATool (Coad and Yourdon 91) supports Coad and Yourdon Object-Oriented Analysis. TurboCASE (StructSoft 92) supports entity-relationship modelling, structured analysis and design methodologies, and object-oriented analysis and design methodologies.

Most CASE tools do not support program implementation. A common approach to assisting implementation is to generate program fragments from a design and allow programmers to incorporate these into their own programs. A major drawback of this approach is the problem of "CASE-gap", where modifications to the design and/or implementation become inconsistent with one another. Thus CASE tools alone do not support evolutionary software development well, as the design, implementation and maintenance aspects of software development are poorly integrated.

CASE tools are often incorporated into software development environments to provide analysis and design capabilities (Reiss 90a and 90b). One problem is that their data storage and user interfaces are difficult to integrate, especially when CASE tools are developed separately from the environment into which they are integrated. A possible solution is to use the FIELD selective broadcasting and user interface "wrapper" approach to tool integration (Reiss 90a). As Reiss notes, however, complete data and user interface integration is usually not possible because of the implicit assumptions CASE tools make about these facilities (providing their own user interface and data storage which is often incompatible with other tools).

2.2.2. Visual Programming

A different use of diagrams to analysis and design is for specifying the execution semantics of a program. This has resulted in the development of "visual" programming languages and environments. Programs are specified using one or more diagrammatic views and these diagrams can be "run" to execute a program. A major advantage of such systems is that they can provide a higher-level, more expressive representation of some program aspects which abstracts away from traditional textual details, such as syntax and linear textual specification (Myers 90, Raeder 85, Ambler and Burnett 89). Many visual programming languages are not entirely pictorial, but use graphical boxes and lines with textual annotations (such as box and line names) to differentiate between diagram components.

PICT (Glinert and Tanimoto 85) uses flowcharts to specify program operations with flowchart boxes containing coloured icons describing the operations they perform. Fabrik (Ingalls et al 88) uses dataflow diagrams to specify user interface behaviour. Dataflow boxes represent various user interface components (such as sliders, buttons and windows) or computational operations (such as addition and sorting). Data "flows" from one box's pins to another's and, as Fabrik programs are always executing, programmers can receive immediate feedback about user interface appearance and behaviour. Prograph (Cox et al 89) also uses dataflow diagrams to specify programs and provides an object-oriented structure where object methods are implemented as dataflow diagrams. An interface builder allows programmers to specify user interface components diagrammatically and specify user interface semantics using dataflow diagrams. These systems are implemented without language or programming environment generators or abstractions and thus a large programming effort is require to define them.

LOGGIE (Backlund et al 90) uses interpreted abstract syntax grammars with "garlands" to provide support for generating graph-based programming languages. It provides attribute grammars based on abstract syntax trees with garlands[1] to provide static semantics checking on directed graphs. Multiple views and graphical representations and editors are supported, including abstract syntax grammar definition using graphical tree representation.

---

[1]A garland is a non-hierarchical link between two abstract syntax tree nodes. This allows graph-based language structures to be represented in conjunction with tree-based abstract syntax structures. The garland approach is a compromise between purely tree-based, hierarchical abstract syntax structures and graph-based program structures, as used by (Arefi et al 90), and is claimed to support more efficient attribute grammar recalculation based on abstract syntax structure (Backlund et al 90).

Deterministic graph transformation systems (Arefi et al 90) use directed graphs to represent programs (nodes are program elements while labelled edges relate program elements). Programs are synthesised by successively applying graph transformations to a program state. Alternative states for the program are defined with generic graph representations and, as long as a graph transformation is deterministic, the original program graph can be transformed from one form to another to visually construct a program. No direct support for language semantics is currently modelled and no multiple view support is provided. The editing mechanism for these program graphs is currently structure-oriented, but allows programs to be altered through a series of non-syntactically correct states alleviating some structure-oriented editor restrictions (Arefi et al 90).

GARDEN (Reiss 86 and 87) provides an environment for prototyping visual programming languages and for conceptual programming with several different languages. All data is represented by objects which provide a structural (syntactic) language representation scheme. These objects also provide support for both static and dynamic language semantics. Views are defined as dependencies between objects moderated by a third object. GARDEN uses an object-oriented database for program storage and for uniform tool data storage. The internal structure of objects can be edited using text as well as by using graphical editors on multiple object representations. New environments are implemented by reusing pre-defined GARDEN objects and tools. Reiss notes that GARDEN is useful for visual language prototyping but integration with existing tools is difficult (Reiss 86 and 90a).

Vampire (McIntyre and Glinert 92) is a set of tools for visually constructing iconic programming environments. Vampire supports the construction of new visual programming environments using a form of visual programming. This allows new visual environments to be built much more easily than using a textual language with user interface libraries or toolkits. There is currently no support in Vampire for building textual view editors or multi-view editing environments with graphical/textual view consistency.

GraphLog (Consens and Mendelzon 92) provides a querying and graph visualisation system built using a concept of Hygraphs. GraphLog supports querying and visualisation of both database schemas and data instances using Hygraphs. Queries are visually constructed and results automatically laid out according to the form of a graphical query. Hy+, a successor to GraphLog, provides improved querying facilities and allows textual retrieval and update via Hygraph queries (Consens and Mendelzon 93). This text can be kept consistent with graph updates in a rudimentary way.

GLIDE (Kleyn and Browne 93) provides a grammar for specifying the structure and semantics of graph-based visual languages. New environments for these languages are generated from a grammar specification and the EDGE graph editor (Paulisch and Tichy 90) is used to implement these environments. GLIDE uses structure-oriented editing of

graphs but supports translation of a program graph from one syntactically correct state to another via one or more non-syntactically correct states (similar to deterministic graph transformation systems (Arefi et al 90)). GLIDE also supports dynamic semantics specification, simple dynamic visualisation and multiple views of a program (which can't be directly edited).

Some programmers find using purely visual representations of a program unwieldy for constructing and visualising some control statements and, in particular, expressions (Ratcliffe et al 92, Myers 90, Vlissides 90). Some systems try to over-come these problems with expression evaluators (where expressions are described using text). A further problem is lack of formal definitions for visual language syntax (i.e. what a picture actually means) (Golin and Reiss 90), which has made generation of visual language environments difficult.

2.2.3. Program Visualisation

Program visualisation systems use graphical program representations to describe program data and execution states in a high-level manner (Myers 90, Ambler and Burnett 89, Brown 88). Such systems can be used to debug programs by showing low-level data and execution states, describe how a software system works by showing relationships between software components, and animate algorithms by showing changes to data and execution flow. Static program visualisation describes the structure (and possibly control-flow) of a program specification. Dynamic program visualisation describes the data and execution-flow of a running program.

Many visual programming systems use the program construction diagrams to visualise a program running. Examples include Fabrik and Prograph which allow programmers to visualise executing dataflow programs in a similar form to their dataflow specifications. Pins and boxes representing an executing program have data values associated with them which programmers can view to determine if a program is executing correctly. This provides a more powerful and easier to use debugging interface than conventional text-based debuggers, such as the THINK C debugger (Symantec 89) and Unix dbx debugger (even with a graphical user interface (Reiss 90b)). Programmers can "see" data move between dataflow boxes and can move between and use debugging views more easily and naturally than textual data and control-flow displays (Myers 90).

GraphTrace (Kleyn and Gingrich 88) records message dispatches for an object-oriented language and uses these to produce an animation of the running program. This aids in the understanding of the program's structure and how the program works. (Haarslev and Möller 90) describe a system for specifying program visualisations using a T$_E$X-like description language. CLOS programs can be visualised both statically and dynamically using this language extension.

BALSA-II (Brown 88) can be used to animate low-level program control-flow and to describe high-level algorithm animation. Programs are written in a language defined by BALSA-II and a pre-defined set of program visualisation views are provided. The main disadvantage of BALSA-II is that new animation views must be written using low-level C code and toolkit routines. In addition, BALSA-II is unsuitable for animating programs not written using its internal language.

TANGO (Stasko 89) provides similar animation capabilities to BALSA-II but allows animations to be specified much more abstractly and for a wider range of programming languages. Animation views can be specified using a textual specification language (which generates the C code to perform the animation) or using diagrams which are then translated into the textual specification language. Programs are annotated by adding procedure calls to the TANGO animation system at appropriate places, or by using an "annotating editor" supplied by the FIELD environment (Stasko 89, Reiss 90b).

Zeus (Brown 91) supports data and code visualisation by adding "event generators" to procedure calls for Modula-2 programs. A pre-processor adds event generators to all procedure calls and views are constructed which make use of these events. Zeus provides facilities for quite sophisticated program visualisation and algorithm animation, including the use of sound and colour. Zeus views are implemented in Modula-2 by reusing a set of pre-defined procedure calls.

A general problem with most program visualisation systems is how to abstractly specify the visualisation or animation required. For software development environments, programmers typically require tools to help statically visualise complex software and to assist in debugging complex software (Reiss 90b, Brooks 87). This does not usually require complex animations, which are difficult to specify abstractly, as provided by systems such as TANGO and BALSA-II. For most static program visualisation, an ability to construct new diagrams, possibly from information already defined about a program, is a general requirement (Reiss 90b, Kleyn and Gingrich 88). A level of dynamic visualisation suitable for debugging programs and visualising data structures is usually sufficient for most programming tasks (Haarslev and Möller 90).

2.3. Integrated Software Development Environments

As textual and graphical program representations are both useful, integrating the two may be a good approach to integrating different phases of software development (Meyers 91, Ratcliffe et al 92). This integration should solve the problem of inconsistent design and implementation by providing consistency management between different views of software development. Another advantage is that diagrams useful for one phase of development can be used in another phase. For example, an analysis-level class hierarchy

is useful for browsing class interface implementations (Fischer 87) and design diagrams are useful for accessing code modules (Wasserman and Pircher 87).

In addition, other environment tools could be usefully reused to avoid tool redundancy (Meyers 91). For example, version control tools are useful for program code version and configuration management, but may also be useful for design and analysis diagram version control. A design may be evolutionary, particularly for object-oriented software development (Henderson-Sellers and Edwards 90). During development and maintenance, different aspects of large software systems may need to be re-analysed and re-designed, requiring multiple versions of analysis, design and implementation views.

PECAN (Reiss 85) provides an integrated environment for Pascal programming using multiple textual and graphical representations of a common program. PECAN provides a program representation and semantics calculation model based on trees. Multiple graphical and textual views are supported with graphical views using a structure-edited approach while textual views use an incremental parsing algorithm. PECAN does not support version control but does have a flexible undo/redo facility which includes macro-operations. Kaiser observes that the PECAN model would be difficult for most people other than its designers to reuse, due to its complexity (Kaiser 85).

FIELD environments (Reiss 90a and 90b) provide the appearance of an integrated programming environment built on top of distinct Unix tools. Program representation is usually as text files with each tool supporting its own semantics (with conventional compilers and debuggers). Views are not directly supported but tool communication via selective broadcasting (Reiss 90a) allows changes in one tool "view" (for example, an editor) to be sent to another tool "view" (for example, the debugger or compiler). Free-edited textual program views are supported (but these text views cannot contain over-lapping information) while graphical representations are generated from cross-reference information. Reiss notes that a lack of user-defined layout and view composition for these graphical views is a problem (Reiss 90b). Version control is not currently supported (although it is planned). Data storage is via Unix text files and a simple relational database (for cross-reference information). Data integration is thus not directly supported but tools can implement translators, driven by selective broadcasting, to allow data from one tool to be used by another. A common user interface "look and feel" is supported by providing a graphical user interface front-end for Unix tools.

Ispel (Grundy et al 91) provides a generic visual programming environment for object-oriented languages. Ispel provides multiple diagrams supporting the construction of inheritance and aggregation relationships between classes which are edited to specify an object-oriented program. Textual views of a class can be generated from these diagrams but can not be edited. Ispel is implemented without reusing abstractions for program representation or view consistency.

Dora (Ratcliffe et al 92) environments support multiple textual and graphical views of software development with all editing structure-oriented. Dora uses the Portable Common Tool Environment (PCTE) (Wang et al 92) to store program data and uses PCTE view schemas to provide selective tool interfaces to these programs. Dora tools are specified using an Editor Description Language and are implemented using the Interviews (Linton et al 88) user interface toolkit. Dora supports the construction of analysis and design views, as well as implementation code views, but assumes these views are updated by structure-oriented editing of base program data. It is not clear what the effect on an abstract, design-level structure is when a corresponding code-level structure is updated.

The distributed object-oriented programming environment of (Nascimento and Dollimore 93) provides a distributed, multi-user programming environment for Smalltalk. Currently, only existing Smalltalk tools are supported (mostly text-based) but multiple users can work on the same program at one time. Programs are distributed with a shared program store representing the definitive state of a program. Programmers can have a different version of Smalltalk classes and versions can be merged by the "owner" of a class as necessary. Tool data integration is via the distributed object space supported by a distributed Smalltalk used to implement the environment.

2.4. General Requirements

From the example systems in the previous sections we can discern a general trend in the development of ISDEs. These form the basis of a general set of requirements for ISDEs:

- Both textual and graphical representations of software are useful. Supporting both representations provides software developers with a choice of representation appropriate to a given development task. Editing style should be flexible and appropriate to the kind of representation used.
- Multiple views of software development must be kept consistent to ensure developers do not mis-understand the current state of a system and do not make inconsistent modifications.
- Multiple tools are useful for software development and these tools need to be integrated to provide a consistent user interface and share access to the same data. An ISDE should be extensible in that new tools can be interfaced to the environment (both data and user interface) in a consistent manner without affecting the performance of existing tools.
- Development of ISDEs is a large programming effort and appropriate abstractions for constructing such environments are very useful. A model for ISDEs should include: flexible program structure and semantics representation; support for definition of different views and view representations; view consistency management; editor functionality and user interface specification; and tool integration mechanisms. A common set of building-blocks should also

be provided, based on this ISDE model, so new environments and tools can be constructed which reuse these environment abstractions.

In the following chapters, these requirements are met by a new model for ISDEs. Chapter 3 introduces Snart, a simple object-oriented language used as an example language to produce an environment for. Chapter 4 describes an ISDE for Snart which fulfils the requirements discussed above. Chapter 5 describes the MViews model for ISDEs and two languages used to specify MViews-based environments. Chapter 6 provides an object-oriented architecture for designing MViews-based ISDEs and Chapter 7 describes an implementation of this architecture as a framework of Snart classes. Chapter 8 demonstrates how SPE, and a generalisation of SPE, can be designed and implemented using the MViews architecture and Snart framework. Chapter 9 further demonstrates the reusability of MViews by developing several other environments.

Chapter 3

Object-oriented Programming in Prolog with Snart

In this chapter we discuss Snart, a set of object-oriented extensions to Prolog. Snart was developed during this research to provide a simple, representative object-oriented programming language for illustrating concepts and implementing the systems we have designed. Snart provides the basic facilities found in many object-oriented languages, together with dynamic classification, a facility previously only provided by Kea (Hamer 90, Hosking et al 90, Hamer et al 92).

The rationale for Snart and its facilities are discussed in the context of a simple program implemented using the language. Software development in Snart is described, and the language compared to other object-oriented Prologs. We briefly comment on the language's performance, classification in Snart and future extensions we envisage. A detailed description of Snart, its environment, and its implementation is provided in Appendix B.

3.1. Object-oriented Programming

We focus on object-oriented languages, their programming environments and software engineering techniques and tools for these languages for three main reasons:

- As software applications get ever larger, better software engineering techniques and methodologies must be employed to manage the growing complexity of problems (Meyer 88, Henderson-Sellers and Edwards 90, Monarchi and Puhr 92). Object-oriented analysis, design and implementation can assist these management tasks (Meyer 87, Meyer 88, Henderson-Sellers and Edwards 90), and thus we expect such languages and techniques to gain a growing following.

- Programmers require tools that assist them with software construction by allowing appropriate use of abstraction, selective views of software development, and help manage change to complex software (Dart et al 87, Henderson and Notkin 87). Recent developments and experience with Object-Oriented Analysis (OOA) and Design (OOD), and complementary tools for these methodologies, suggest object-oriented modelling techniques and

languages can help produce higher quality software than conventional techniques (Fichman and Kemerer 92, Monarchi and Puhr 92).

- Object-oriented languages are, by their object-based focus, appropriate candidates for both visual programming and program visualisation. They also provide a natural method of describing and implementing the models developed in this research.

The reader is assumed to have a good working knowledge of most object-oriented concepts. These include object-oriented analysis and design (Coad and Yourdon 91, Henderson-Sellers and Edwards 90), object-oriented languages (Meyer 88, Winblad et al 90, Stroustrup 86), some implementation issues with object-oriented languages and systems (Goldberg and Robson 84), and an appreciation of the differences between abstract data types (Meyer 88), meta-classes (Goldberg and Robson 84) and prototypes (Lieberman 86, Ungar et al 92).

3.2.   Rationale for Snart

The Snart language was developed to provide a simple object-oriented language for different phases of our research. We required a programming language for implementing the systems we developed that provided:

- The flexibility of Prolog for experimental rapid prototyping. Prolog was used to good effect in the development of Ispel (Grundy et al 91) and we wished to make further use of the language's prototyping facilities. Features useful for experimental programming include: modification of source code while retaining run-time data; a high-level, declarative style that can be easily modified to accommodate design changes; and a language structure that isn't greatly affected by design modification (for example, limited type checking).
- Object-oriented structuring for programs, as opposed to less flexible and reusable conventional Prolog, C or Pascal program structure.
- Integration with existing programs and libraries, including parsing support, graphical user interface construction support, and database support. Of particular importance was good graphical user interface support, such as that provided by the LPA MacProlog system (LPA 89).
- A language with sufficient run-time speed for interactive applications. Access to the compiler and run-time system was required so we could modify and experiment with the language, for example to add explicit classification and run-time object method tracing.
- An integrated development environment including fast compilation, editing, and browser and debugger support to facilitate fast, experimental programming.

We also required a representative object-oriented language as an example language to develop an environment for. This language should be based on common object-oriented principles suitable for software engineering (classes and strong typing versus meta-classes or prototypes), similar to popular languages such as C++ or Eiffel. This should allow us to apply our environments, models and program design reasonably easily to another language, without using less well-supported facilities such as meta-classes, classes-as-objects and prototypes.

In addition to Snart, we needed a Prolog system as an implementation platform. We chose LPA MacProlog due to its rapid prototyping facilities, including incremental compilation, high-level access to the Macintosh graphics and a WIMPS environment that assists the rapid development of experimental software. A major advantage of using Snart is access to LPA Prolog's high-level, declarative facilities for building graphical user interfaces. LPA provides a declarative Graphics Description Language (GDL) for specifying a wide range of graphical *pictures* (for example, boxes, lines, ovals, text, shading, and composite objects). LPA also provides many predicates for manipulating these pictures and for building *graphic windows* (editable windows containing pictures).

In addition to graphical picture manipulation, LPA provides high-level access to the Macintosh windowing system and user interface facilities. Menus, dialogues and windows are all specified in a declarative style with one Prolog predicate call often being sufficient. Although LPA's facilities are implemented in Prolog and don't have an object-oriented structure, Snart classes can be defined to interface to them and provide a user interface framework similar to Interviews (Linton et al 88), the THINK Class Library (Symantec 91), and GARNET (Myers 90). Appendix A gives a brief overview of the facilities of LPA for building graphical user interfaces.

Extending the LPA language and environment to provide facilities for Snart programming gave us a good development system for implementing the results of our research. It also provided a motivation for replacing our simple Snart environment with a visual programming environment supporting design, implementation and maintenance phases of software development. This environment and its implementation could then be compared to development without it.

3.3. A Snart Example: A Drawing Program

In this section we describe Snart by example by showing how Snart can be used to implement a simple drawing program using the graphical facilities of LPA and the Macintosh. Fig. 3.1. shows a screen dump from this program.
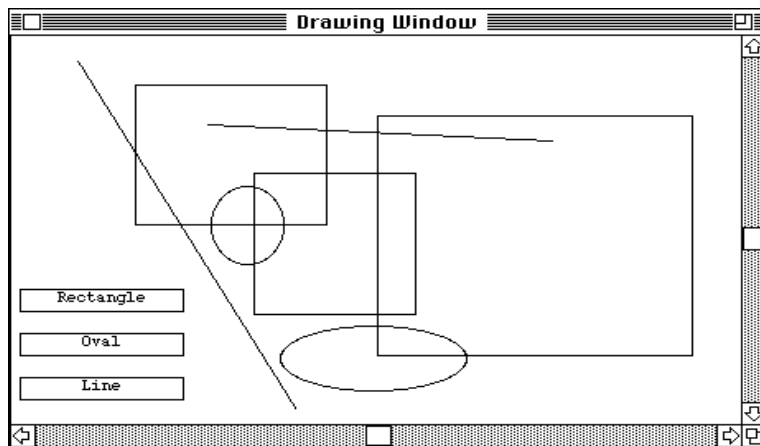
fig. 3.1. Screen dump from a simple drawing program implemented using Snart.

3.3.1. Snart Classes and Methods

The drawing program is composed of classes that implement a drawing window, buttons that select a type of figure to be drawn and different figures that can be drawn in the window. The `figure` and `button` classes form a hierarchy. Fig. 3.2. shows the Snart code that implements the `figure` class of the drawing program.

```
% The figure class.                    % Delete a figure
% All figures inherit from this class.  %
%                                       figure::delete(Figure) :-
abstract_class(figure,                   Figure@hide,
 parents([]),                            Figure@window(Window),
 features([                              Window@remove_figure(Figure),
  window:drawing_window,                 Figure@dispose.
  location:term,
  visible:boolean,                      % Hide a figure.
  frame:term,                           %
  create,                               figure::hide(Figure) :-
  draw(deferred),                        Figure@window(Window),
  hide,                                  Window@del_pic(Figure),
  resize(deferred),                      Fig@visible:=false.
  delete,
  info,                                 % Information on this figure
  pt_in_figure                          %
 ])).                                   figure::info(Figure) :-
                                         Figure@location(Location),
% Creation method for a figure.          writeseqnl(['Location =',Location).
%
figure::create(Figure,Window,Loc) :-    % See if given point is in this figure
 Figure@window:=Window,                 %
 Figure@location:=Loc,                  figure::pt_in_figure(Figure,X,Y) :-
 Figure@visible:=false,                  Figure@visible(true),
 Window@add_figure(Figure).              Figure@frame(Frame),
                                         pt_in_box((Y,X),Frame).
```

fig. 3.2. The *figure* class.

The `abstract_class` predicate defines a class called `figure`, with no parent classes it inherits from; attributes `window`, `location`, `visible`, and `frame`; and methods `create`, `draw`, `hide`, `resize`, `delete`, `info`, and `pt_in_figure`. Method definitions are of the form `ClassName::MethodName(ObjectID,Argument`$_1$`,...,Argument`$_n$`):-Body`, where ObjectID is

the object ID the method is invoked for (i.e. the object a "message" is sent to). The method implementation is the same form as standard Prolog clause bodies.

Features of an object are referred to by `ObjectID@FeatureName`. Methods are invoked by `ObjectID@MethodName(Argument_1,...,Argument_n)`. Attributes are assigned values by `ObjectID@FeatureName:=Value`, and fetched by `ObjectID@AttributeName(Value)`. Note the method invocation and attribute fetch syntax are identical, as in Eiffel (and thus behave as "feature calls").

Methods with no implementation in `figure` are deferred for implementation by a subclass. These "features" can be implemented by either attributes or methods in subclasses. The "types" associated with attributes in a class definition are not used for compile-time or run-time type checking. These are used to default values for attributes and form a valuable documentation aid. A visual programming environment can use such declarations to build its own data structures from Snart code (see Chapter 4). Types can be term, atom, integer, string, list, or boolean (i.e. standard Prolog types), or Snart class names (thus a class can be an aggregate of other class types).

Fig. 3.3. shows the Snart code that implements the rectangle class of the drawing program.

```
% The rectangle figure class.              % Draw a rectangle figure
%                                           %
class(rectangle,                            rectangle::draw(Rect) :-
 parents([                                   Rect@window(Window),
   closed_figure(                            Rect@location((X,Y)),
    [rename(create,fig_create),              Rect@width(Width),
       rename(info,closed_info)])            Rect@height(Height),
 ]),                                         Window@add_pic(Rect,
 features([                                   box(Y,X,Height,Width)),
  height:integer,                            Rect@visible:=true,
  width:integer,                             Rect@frame:=box(Y,X,Height,Width).
  create,
  draw,                                     % Area for a rectangle
  resize,                                   %
  area,                                     rectangle::area(Rect,Area) :-
  perimeter,                                 Area is Rect@width * Rect@height.
  info
])).                                        % Perimeter for a rectangle
                                            %
% Create a rectangle figure                 rectangle::perimeter(Rect,Perimeter) :-
%                                            Perimeter is 2 *
rectangle::create(Rect,Window,               (Rect@width + Rect@height).
  Location,Width,Height) :-
 Rect@width:=Width,                         % Resize a rectangle
 Rect@height:=Height,                       %
 Rect@fig_create(Window,Location).          rectangle::resize(Rect,NX,NY) :-
                                             Rect@width:=NX,
% Information on this rectangle              Rect@height:=NY,
%                                            Rect@draw.
rectangle::info(Rect) :-
 writenl('Information for rectangle:'),
 Rect@closed_info.
```

fig. 3.3. The *rectangle* class from the drawing program.

Concrete class `rectangle` has `closed_figure` as a parent, which in turn inherits from `figure`. The `create`, `info`, `draw`, and `resize` methods have been overridden in `rectangle`,

and the attributes `height` and `width` added. The `figure::create` and `closed_figure::info` methods are still available to `rectangle` via renaming as `fig_create` and `closed_info` respectively. Snart provides multiple and repeated inheritance, so `rectangle` could also inherit from another class `four_sided_figure`, that provides methods and attributes specific to four-sided figures.

Note the interaction between Snart methods and standard Prolog code. `rectangle::info` calls `writenl` (an LPA Prolog predicate), and the Prolog expression evaluator `is` can call `Rect@height` and `Rect@width` to evaluate the area and perimeter of a rectangle. Predicates can not call Snart methods directly, but invoke them via object feature calls, as in the later example.

3.3.2. Snart Objects

Fig. 3.4. shows the `drawing_window` and `rect_button` classes from the drawing program. When a user clicks the mouse inside the drawing window, and drags a marqui shape[2], a new figure is added. The `current_button` attribute of `drawing_window` determines the button object to call to create this new figure. The `rect_button::add_figure` method adds a new rectangle figure by calling `Rect::create(rectangle,...)`. `create` is a "distinguished" method[3], which creates a new object of type given by the first argument, in an analogous fashion to the Eiffel `Create` routine. To destroy an object, the distinguished method `ObjectID@dispose` is called (see `figure::delete` in fig. 3.2.). Further distinguished methods exist for various object manipulation tasks. Appendix A describes these in detail.

---

[2]A marqui shape is an outline of some graphical figure which is interactively resized around a fixed point. A typical example of marqui use is when selecting a group of icons in a window. Users typically drag a box-shaped marqui around the icons they wish to select and these icons are the selected when the mouse button is released. The drawing editor program uses box and oval marquis to allow users to interactively determine the size of a new figure.

[3]A distinguished method is one supported by all classes of objects. For example, `create`, `copy` and `dispose` are some of distinguished methods supported by Snart (see Appendix B for a full list of these distinguished methods).

```
% The drawing_window class                 % Add a figure to the figure list
%                                           % of this window.
class(drawing_window,                       %
 parents([                                  drawing_window::add_figure(Win,Figure) :-
  window([                                   Win@figures(Figures),
    rename(clicked,window_clicked)])         Figure@info,
 ]),                                         Win@figures:=[Figure|Figures].
 features([                                 drawing_window::add_figure(Win,Figure) :-
  buttons:list(drawing_button),              Figure@info,
  current_button:drawing_button,             Win@figures:=[Figure].
  figures:list(figure),
  clicked,                                  % Remove a figure from the figure list
  shift_clicked,                            % of this window.
  add_figure,                               %
  remove_figure                             drawing_window::remove_figure(Win,Fig) :-
 ])).                                        Fig@info,
                                             Win@figures(Figures),
% Process click for drawing window           remove(Fig,Figures,NewFigures),
%                                            Win@figures:=NewFigures.
drawing_window::clicked(Window,X,Y) :-
 % see if button clicked                    % The rect_button class
 Window@window_clicked(X,Y).                %
drawing_window::clicked(Window,X,Y) :-      class(rect_button,
 % draw a figure using a marqui             parents([drawing_button]),
 mouse_down,                                  features([
 Window@lpa_window(Name),                     marqui_shape,
 Window@current_button(Button),              add_figure
 Button@marqui_shape(Shape),                 ])).
 marqui(Name,(Y,X),box(T,L,D,W),Shape),
 Button@add_figure(T,L,D,W).                % Marqui shape for rectangle is "box":
drawing_window::clicked(Window,X,Y).        %
 % default - do nothing                     rect_button::marqui_shape(Button,box).

% Process shift-click for drawing window    % Add a rectangle figure to window.
% (deletion of a figure)                    %
%                                           rect_button::add_figure(Button,T,L,D,W) :-
drawing_window::shift_clicked(Win,X,Y) :-    Button@window(Window),
 Win@figures(Figures),                       Rect@create(rectangle,Window,(L,T),W,D),
 on(Figure,Figures),                         Rect@draw.
 Figure@pt_in_figure(X,Y),
 Figure@delete.
drawing_window::shift_clicked(Win,X,Y).
```

fig. 3.4. The *drawing_window* and *rect_button* classes.

3.4.   Classification in Snart

The Kea language developed at the University of Auckland (Hosking et al 90) supports a novel, strongly-typed object classification facility (Hamer et al 92). Objects of a general type can be created, and then specialised to sub-classes at run-time, as more information about the object is obtained. Kea is a lazy, functional language, and this classification process takes place lazily as required.

Snart provides a form of classification in an imperative language setting. Classes can have one or more classifier attributes (for example, the `shape` attribute of `figure` in fig. 3.5), whose types are a list of sub-types a class can be classified to. For example, the `classifiable_figure` class in fig. 6 has one classifier `shape` which indicates `classifiable_figure` can be dynamically specialised to `rectangle`, `foval` or `fline`.

To classify a `classifiable_figure` object at run-time:

```
        ObjectID@classify(shape,rectangle)
```

will change the object's class to be `rectangle`. A classification using `classify` is valid if:

- the classifier attribute exists
- the given class to classify to exists and is on the classifier's class list
- the given class is a sub-class of the class of the object.

An object can be re-classified to another class (a member the classier attribute's list) and any attribute values incompatiable with its new class are removed.

```
% The figure class (with classifier)
%
class(classifiable_figure,
 parents([figure]),
 features([
  shape([rectangle,foval,fline]),
  draw,
  resize
 ])).
% Create a new abstract figure object
%
drawing_window::create_figure(Window,
  Location,Figure) :-
 Figure@create(classifiable_figure,
  Window,Location).
% Classify figure to a rectangle figure
%
drawing_window::figure_rectangle(Window,
  Figure,Width,Height) :-
 Figure@classify(shape,rectangle),
 Figure@width:=Width,
 Figure@height:=Height,
 Figure@draw.

% Classify figure to an oval figure
%
drawing_window::figure_oval(Window,Figure,
  Vertical,Horizontal) :-
 Figure@classify(shape,foval),

 Figure@v_radius:=Vertical,
 Figure@h_radius:=Horizontal,
 Figure@draw.

% Change oval figure to rectangle figure
% using classification
%
drawing_window::oval_to_rectangle(Window,
  Oval) :-
 Oval@v_radius(Height),
 Oval@h_radius(Width),
 Oval@classify(shape,rectangle),
 Oval@height:=Height,
 Oval@width:=Width,
 Oval@draw.

% Change rectangle figure to oval figure
% using classification
%
drawing_window::rectangle_to_oval(Window,
  Rect) :-
 Rect@height(Vertical),
 Rect@width(Horizontal),
 Rect@classify(shape,foval),
 Rect@v_radius:=Vertical,
 Rect@h_radius:=Horizontal,
 Rect@draw.
```

fig. 3.5.   Using classification to specialise and re-classify figures.

A class may have multiple classifiers. An object may then be classified once using one classifier and then again using another classifier. The affect of this is to produce an object whose class is a "merging" of the two classes it has been classified to. For example, if `classifiable_figure` had an additional classifier `visibility : [hidden,foreground,background]`, then an object of this class could be classified first to `rectangle` and then to `hidden`.

Fig. 3.5. shows an example using classification. An abstract figure can be created and initialised, and later specialised to a specific figure type. Figures created in this manner can also be re-classified to other shapes. Note that no objects are created or destroyed, hence all references to the classified objects remain unchanged.

3.5. Object Tracing and Persistency

In addition to classifiers, Snart provides two other object management facilities. Object spying allows run-time Snart objects to be selectively "spied" to produce events equating

to object attribute updates and method entry/exit (which provides an object tracing mechanism similar to those of (Noble and Groves 91) and (Brown 91)). Object persistency allows objects to be made persistent in a global "object store", in a similar manner to the persistent CLOS objects of (Attardi et al 89).

3.5.1. Object Tracing

A Snart object can be spied by specifying the features of the object's class that should generate events or by simply generating events for all features of an object. For example, to indicate an object `RectID` of class `rectangle` should produce spy events the following predicate calls are used:

```
sn_trace_object(RectID)
    % to spy all figure class features for RectID
sn_trace_object(RectID,[draw,location,resize])
    % to spy draw and resize methods and location attribute
```

When an object has spied methods invoked or spied attributes changed, events are generated by calling the user-defined predicates `sn_entry`, `sn_exit` and `sn_set_value`. For example, if the `RectID` object has been spied as above, events of the form:

```
sn_entry(RectID,draw)
sn_exit(RectID,resize(20,30),true)
sn_set_value(RectID,location,(50,75))
```

might be generated when executing the drawing program. Defining appropriate handlers for the event methods allows programmers to handle such object events and to take appropriate action (for example, creating a list of calls to a method or animating a view of the object's state).

This object spying mechanism is useful for keeping run-time objects consistent with views of the object. Chapter 9 illustrates the use of this facility of Snart in a simple dynamic program visualisation system.

3.5.2. Object Persistency

Snart objects can be made persistent by creating or opening an *object store* which records the state of all persistent objects for a Snart program. A class inheriting from special `persistent` class will have all of its instances made persistent using the currently open object store. Reopening an object store extends Snart's object space to incorporate objects within the store. For example, the following commands create, open and extend an object store. Reopening the store allows access to the persistent objects it contains. In this example a prectangle class is used, which is defined as `class(prectangle,parents([rectangle,persistent]),features([]))` (i.e. a persistent version of `rectangle`). Fig. 3.6. shows an example session from Snart using a persistent

rectangle class. The rectangle object is created then the object store closed. After reopening the object store, the previously created rectangle still exists and can be queried.

```
?- sn_create_object_store(objects,'...path name...')
yes
?- sn_open_object_store(objects,'...path name...')
yes
?- Rect@create(prectangle,Window,(10,10),20,30)
Rect=1
yes
?- sn_close_object_store(objects)
yes
?- sn_open_object_store(objects,'...path name...')
yes
?- 1@info
Location: 10, 10
Perimeter: 60
Area: 200
yes
```

fig. 3.6.   Persistent objects in Snart.

Only updated objects are rewritten to the store when it is closed and objects are not loaded into memory until they are required. Persistent objects that have not been updated can be automatically purged from memory if required. Currently Snart does not permit more than one object store to be used at a time nor does it support merging of object stores.

3.5.3.   Implementation

Both object spying and object persistency are implemented by "classifying" Snart objects to new classes incorporating the spying or persistency maintenance required by the object. Default meta-level methods, such as `sn_set_value`, `sn_despatch_method`, and `sn_alloc_id`, are defined for all Snart classes. The behaviour of objects can be changed by classifying objects to `persistent` and `spy` classes which over-ride these meta-level methods. `spy` defines new attribute setting and method calling behaviour to generate tracing events. `persistent` defines new object allocation, manipulation and destruction methods to support persistent objects. Appendix B explains this meta-level use of object classification mechanism in further detail.

Object spying is used for visualizing Snart programs in Chapter 9. Object persistency can be used to make Snart programs persistent, as described in Chapters 7 and 9. Chapter 10 discusses an extended form of Snart persistent objects for abstract and natural persistency mechanisms for programming environments.

3.6. Software Development in Snart

Snart offers several advantages over developing software using only conventional LPA Prolog:

- *Program structuring.* Classes are used to structure a program in an object-oriented manner. Conventional Prolog programs can be called from Snart or can make use of Snart programs. This allows more high-level, abstract program structuring techniques than those provided by LPA Prolog alone, with both the data and functionality of programs encapsulated in classes.

- *Uniform data storage.* Data is stored as Snart objects rather than using assert and retract or directly using LPA property management. This provides both object-oriented access and modification of data and there is less impact on a program when classes are modified than if asserted terms are modified. Compile-time and run-time checking provide improved checks on the integretity of Snart data as opposed to database or property data (though not as complete as for strongly typed languages such as Kea or Eiffel).

- *Stability under data restructuring.* Snart objects can still be accessed after their class definitions are modified, whereas Prolog database clauses are often inconsistent after predicates using them are modified. Direct access to LPA property management can cause difficulties when trying to change property or object names or when deleting properties, both of which are handled automatically by Snart. Due to this flexibility, Snart supports rapid prototyping with a changing design better than raw LPA Prolog.

- *Support for reuse and frameworks.* Snart classes can be specialised and their behaviour modified, including supporting multiple and repeated inheritance with renaming of features. Generics and parameterised classes can be indirectly supported as Snart has no concept of types associated with object referencing variables. These facilities greatly extend the reusability of standard LPA Prolog predicates.

- *Hybrid language programming.* Snart is similar to C++ in that it is a hybrid language. Snart supports declarative logic programming inside an imperative object-oriented programming structure.

In summary, Snart provides a good rapid prototyping language supporting both the object-oriented and declarative programming paradigms. Chapter 7 discusses our experience of implementing the major components of our research in Snart.

3.7. Other Object-Oriented Prologs

We briefly compare Snart to other object-oriented Prologs, including Protalk (Quintus 91b), Prolog++ (Pountain 90) and ObjVProlog (Malenfant et al 89). Snart was developed in preference to using an existing object-oriented Prolog as:

- We required a simple language with representative facilities found in most object-oriented languages. Many object-oriented Prologs tend to take an approach that is somewhat incompatible with strongly-typed languages such as Eiffel and C++.
- We have complete control over the syntax, semantics and implementation of Snart. The language has already been used as part of a program visualisation project (Fenwick and Hosking 93) which required the method despatcher to be altered to generate events. We anticipated such modifications would be required in our research and might be difficult to implement using existing object-oriented Prologs.
- Snart has a very simple, clean syntax that integrates well with conventional Prolog predicates in the LPA environment and a small compiler and run-time system which are simple to understand and modify.
- Due to the similarities of Snart to C++ syntax and semantics, we hope to be able to port Snart programs to C++ without substantial modification of their design and structure. We also hope to apply the results of our research to class-based languages such as C++. Most other object-oriented Prologs are founded on a different conceptual view of classes and objects which is more difficult to compare with languages of interest to us (e.g. Kea, Eiffel and C++).

Protalk, Prolog++ and ObjVProlog all treat classes as objects. Thus new classes are defined by creating instances of a "class" object by calling a `new_class` method for class. Snart treats classes as abstract data types like Eiffel and C++. Classes and methods are defined and compiled with LPA Prolog code and class definitions are compiled in a separate compilation phase. This results in easier definition and maintenance of classes with the LPA Prolog environment than creating class objects as in Protalk and ObjVProlog. Prolog++ classes are compiled to objects when LPA windows are compiled, as with Snart. The Snart class definition and method syntax are somewhat clearer to read than those of Protalk and ObjVProlog.

ObjVProlog provides meta-classes in a similar manner to CLOS (Attardi et al 89). Classes and meta-classes (Goldberg and Robson 84) are both implemented as objects and can be specialised to provide new object-based facilities that co-exist with existing ones. For example, meta-classes can be defined which implement persistent objects (Attardi et al 89), parallel objects and part-whole hierarchies (Malenfant et al 89). As Snart treats classes differently from objects, we provide no meta-level support in Snart, and such facilities

must be implemented as standard classes and objects with extra features to perform these types of specialised processing (which we do for the implementation of MViews as described in Chapter 7).

Protalk, Prolog++ and ObjVProlog all treat attribute access and method calling as distinct kinds of object manipulation. Snart treats them as "feature calls" and allows subclasses to implement deferred features as attributes or methods. This model equates to the Eiffel treatment of classes as implementations of abstract data types (Meyer 88) and is often more flexible and natural than distinguishing between attributes and methods.

Prolog++ defines object (and class) data inside `open_object` and `close_object` predicates. This means extra Prolog predicates not associated with the object (i.e. ones called by the object's methods but also callable from elsewhere) can only be defined outside the object definition. As Prolog++ and Snart are hybrid languages which often make use of conventional Prolog code, this restriction can be unwanted. Prolog++ allows such auxiliary predicates to be made private, but this is not always what is intended or required. Snart follows the C++ approach which allows mixing of object-oriented (Snart) and conventional language (Prolog) predicates.

Prolog++ also provides daemon objects (for event and data-driven programming) and information hiding which are currently not supported in Snart. Protalk is implemented in Quintus Prolog and uses the Prolog database for object data storage. As Prolog++ and Snart use LPA properties, they both run significantly faster than Protalk.

In summary, Snart provides similar object-oriented facilities to Protalk, Prolog++ and ObjVProlog but takes a C++ approach to the treatment of classes and objects. The other Prologs variously provide additional facilities including run-time creation of classes, data-driven programming support using daemons and have compiler optimisations not yet provided by Snart. Snart programs are compatible with a wide range of object-oriented languages, however, including strongly-typed languages more suitable for software engineering (Meyer 87). Thus we have chosen Snart as a representative object-oriented language and a more appropriate language for implementation of our research than other Prologs.

3.8. Future Research

Snart can be extended in many ways. The most useful include:
- Explicit redefinition and export of features, as used in Eiffel. This would inform programmers of class definition errors not currently detected at compile-time which can sometimes be quite difficult to determine at run-time.

- Information hiding including public, private and protected features as supported by C++. Snart currently allows any feature to be accessed externally to a class and any attribute to be changed externally.

- Support for data and event-driven programming by providing either daemon support or Smalltalk-like Model-View relationships (Goldberg 84). When implementing parts of our research these facilities had to be explicitly represented and programmed. Chapters 7 discuss the advantages and disadvantages of language support for multiple views on objects.

- Improved compile-time optimisations of Snart programs. This would include direct predicate calling to renamed, inherited features and optimization of method despatching.

- Adding "typed" Prolog variables that reference Snart objects. This would allow a number of checks to be performed at compile-time and more compatibility with C++ and Eiffel. It would also provide more flexibility as variables could be typed (accessing Snart data or built-in Prolog data types), untyped and accessing Snart objects (as now), or standard untyped Prolog variables.

- Lazy, functional evaluation for Snart features similar to that provided by Kea. Snart would then integrate object-oriented, declarative logic and functional programming.

- Further extensions to object persistency to provide multiple object stores, programmer control of which object store to write information to, and improved performance of persistent objects.

Appendix B discusses these extensions in further detail, including how they could be implemented in Snart.

3.9. Summary

We have developed Snart, a set of object-oriented extensions to Prolog. Snart supports multiple, repeated inheritance, arbitrary renaming and redefining of inherited features, typed attributes and untyped method specifications, and integration with standard Prolog predicates. Snart also provides an object classification facility similar to that of Kea, but within an imperative language setting. A simple environment for Snart has been implemented as an extension of the LPA MacProlog programming environment. A much more sophisticated environment supporting multiple textual and graphical views of Snart programs with consistency management has been developed and is described in Chapter 4.

Snart adds object-oriented structuring and data storage capabilities to LPA Prolog which enhances the development of experimental software. Snart views classes as implementations of abstract data types, in a similar manner to C++, Eiffel and Kea. Porting

Snart designs and programs to these languages, and using Snart as a representative of this class of object-oriented programming language, becomes easier than with other object-oriented Prologs that adopt a view of classes as objects.

Chapter 4

The Snart Programming Environment

This chapter presents one of the main products of our research, the Snart Programming Environment (SPE). SPE provides an integrated software development environment for Snart including multiple textual and graphical views of Snart programs with consistency management. SPE is introduced here as it illustrates many of the facilities of software development environments our research aims to support. SPE is also used in the following chapters as an example environment for which different features need to be supported.

The original environment for Snart described in Chapter 3 and Appendix B is very simplistic and this chapter introduces a much more sophisticated environment supporting integrated design, implementation, debugging and maintenance of Snart programs. The rationale behind the SPE is discussed and a user's perspective of developing software in SPE is given. This includes the design, implementation, testing and maintenance of software and facilities for program browsing and managing complexity. We also briefly discuss various extensions that could be made to SPE to further facilitate programming in Snart and similar languages.

4.1. Rationale for Snart Programming Environment

The simplistic environment for Snart described in Chapter 3 provides only a rudimentary extension of LPA MacProlog's programming environment to support the development of Snart programs. The only extra facilities include access to the Snart compiler, location of Snart class definition and methods in program windows, the location and printing of object and class data, and very simple object management and debugging facilities.

As our discussion of programming environments in Chapter 2 noted, many different program construction and visualisation techniques are useful during software development. Some techniques are also useful in other phases of software development. For example, class diagrams are useful for analysis and design (Coad and Yourdon 91, Henderson-Sellers and Edwards 90), during implementation as browsers and for static program visualisation (Haarslev and Möller 90, Symantec 90), and during debugging for object tracing (Kleyn and Gingrich 88, Myers 90).

An integrated environment providing a wide range of program design and construction techniques for object oriented software would ideally provide various facilities as described below:

- Object-oriented software design and implementation are often concurrent or recursive activities (Henderson-Sellers and Edwards 90) with software development being an evolutionary process (Coad and Yourdon 91, Ref). Maintenance and/or enhancement of a software system also causes changes to impact and flow through a system. Thus an integrated environment can propagate changes more easily and automatically than a disjoint system incorporating different, distinct tools (Reiss, 91).

- An essential requirement for environments supporting multiple phases is the need to maintain consistency between the phases. Change to a design must be reflected in its implementation and vice-versa. Most CASE tools generating code or describing an analysis or design, for example Software thru Pictures (Wasserman and Pircher 87), TurboCASE (StructSoft 92), and the OOATool (Coad and Yourdon 91), get out-of-step under design or implementation change. Programmers must manually ensure different aspects of the development are updated and made consistent (or re-generated), a tedious, error-prone and incomplete process. Automation of this process, or tools to help with this automation, as in Ispel (Grundy et al 91), is much more desirable.

- As representation and interaction techniques are useful in more than one phase of development, and at differing levels of abstraction, integrating the phases of development in one environment allows the same or similar techniques and tools to be reused on the same data. For example, a class diagrammer can be used to design a class hierarchy, browse and access class definition code and modify the class hierarchy when extending the design and implementations (Grundy 91).

- Multiple views of information are useful, and to some degree necessary (Monarchi and Puhr 92, Ratcliffe et al 92, Reiss 85, Wang et al 92), to provide programmers with techniques for managing software complexity. Views should:
    - be textual or graphical, as the textual programming paradigm is useful for detail and graphical programming for a high-level overview of programs
    - share information which is kept consistent automatically by the environment
    - be at an appropriate level of detail or abstraction for their task
    - have composition and layout programmer-determined

- Reuse of existing tools where possible and extensibility of the environment is necessary (Reiss 90b). This includes the ability to add new tools and extend old ones.

Ispel (Grundy et al 91) supports some of these concepts although it requires much more versatile program representation and manipulation. We designed and implemented SPE based on many of the concepts of our original Ispel environment to improve Snart programming. In the following sections we describe software development in SPE and illustrate use of the environment with examples designing and implementing the drawing program introduced in Chapter 3.

4.2.   Analysis and Design of a Snart Program

In this section we describe an analysis and design of the drawing program from Chapter 3. We assume development from scratch of a drawing program-like software system and illustrate its design.

4.2.1.   Requirements for the Drawing Program

The drawing program from Chapter 3 provides a window with several buttons (see Section 3.3.). Clicking on a button sets the current "drawing figure" which will be drawn when a marqui is dragged in the window. Dragging a marqui results in the appropriate figure being drawn and information about the figure being displayed (its location, height, width and so on). Shift-clicking on a figure first displays information about the figure and then deletes the figure. Rectangles, ovals and lines are the only three figures initially provided, although extensibility should be aimed for.

4.2.2.   Creating a New Program

The first step when using SPE is to create a new program. A programmer supplies the program name, a cluster name and the name of the first class (clusters are used to group related classes). Fig. 4.1. shows the `root class` view created by SPE when the program `drawing` is created with root class `window`.
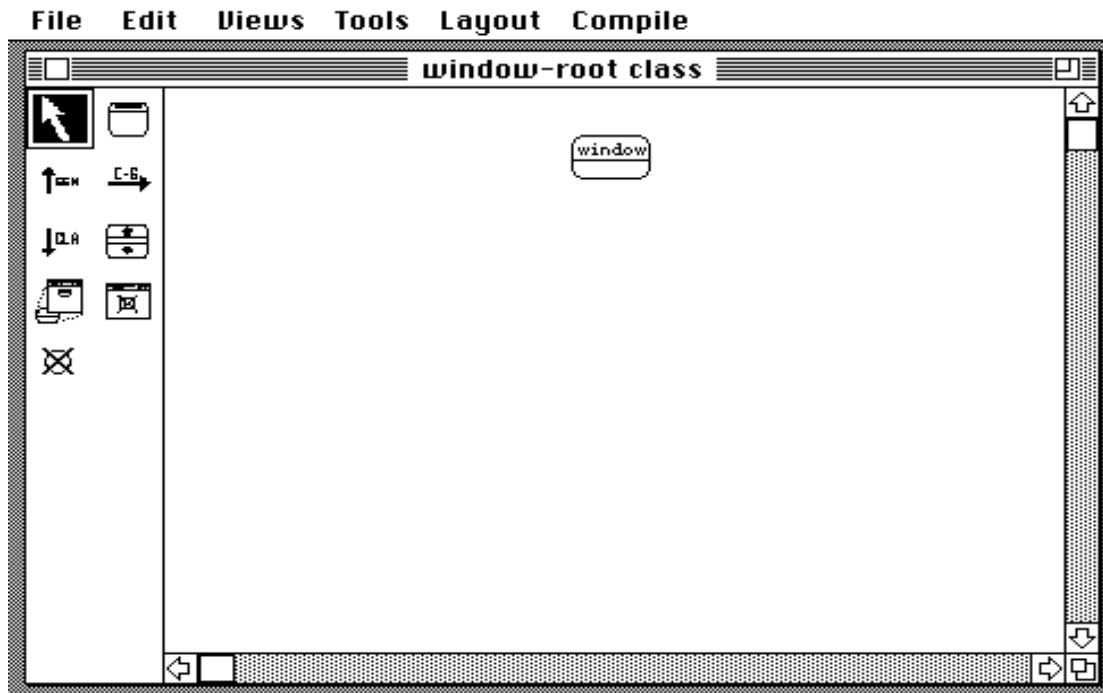
fig. 4.1.　Initial root class view for the drawing program.

SPE class diagram views have a set of tools for manipulating the program associated with their window. Menus control various other aspects of user interaction and are described with associated tools in the following sections.

4.2.3. Analysis

The first step in the analysis of the drawing program is to determine the class hierarchies needed. A special `drawing_window` class is derived from the window class, and the `figure` and `button` hierarchies are defined. Figures can be specialised to `open_figure` and `closed_figure` figures, and buttons to `drawing_button`, using generalisation relationships. Generalisation relationships between classes specify that a class is generalised to one or more other classes, and are typically used for object-oriented analysis. Views are constructed for each hierarchy by selecting the create view tool (⊡) or menu item, clicking on the class icon which will own the new view (be its focus), and giving the new view a name. Classes and generalisation relationships are added to views using the class tool (▯) and generalisation tool (↑ᵉᵐ) respectively.
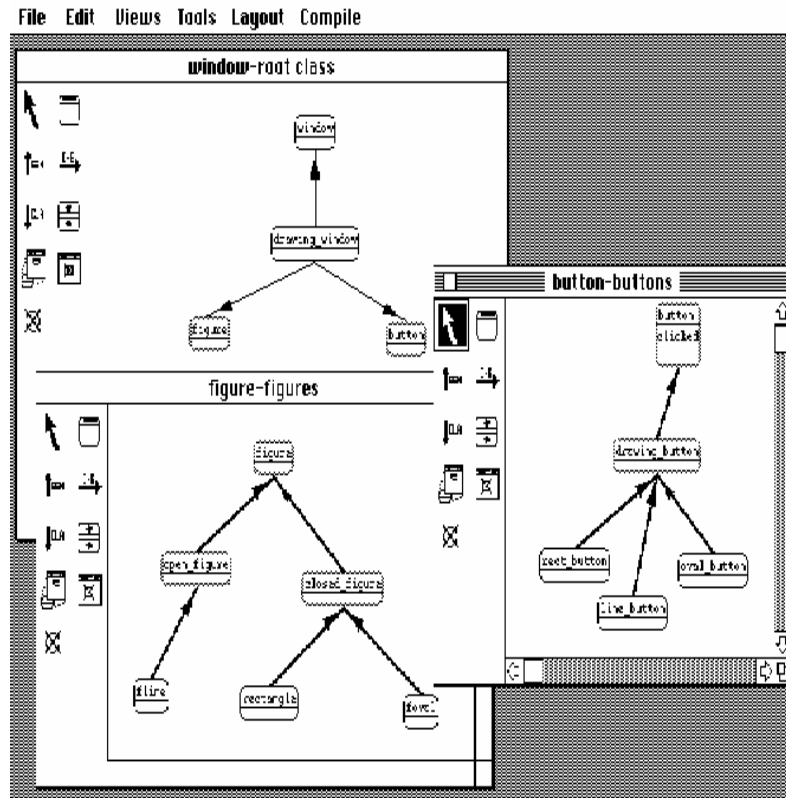
fig. 4.2.   Class hierarchy views for the drawing program.

Important aggregation and association relationships between classes are added using the add client-supplier tool (⟶). A dialogue is used to specify information about the relationship including its name (if any), its arity and whether it is inherited from an ancestor class. An aggregation relationship between two classes indicates that an instance of one class is composed of instances of the other class (i.e. a part-of relationship). For example, a `drawing_window` object may be composed of zero or more `figure` and `button` objects. An association relationship between two classes indicates one class makes use of the features of the other class in some way. For example, a `figure` class may be associated with a `drawing_window` class, indicating the `figure` class uses the `drawing_window` class interface[4] in some manner. Aggregation and association relationships are typically used for object-oriented analysis and are refined into client-supplier relationships (Henderson-Sellers and Edwards 90). Classes can be selected and dragged using the selection tool (➤) in a similar manner to figures in a drawing package.

Fig. 4.2. shows the three views containing each hierarchy. The bold arrowed lines represent generalisation relationships while the thin arrowed lines represent aggregation and association relationships between classes.

---

[4]A class interface is the set of attributes and methods defined by the class and inherited from a class's generalisations.
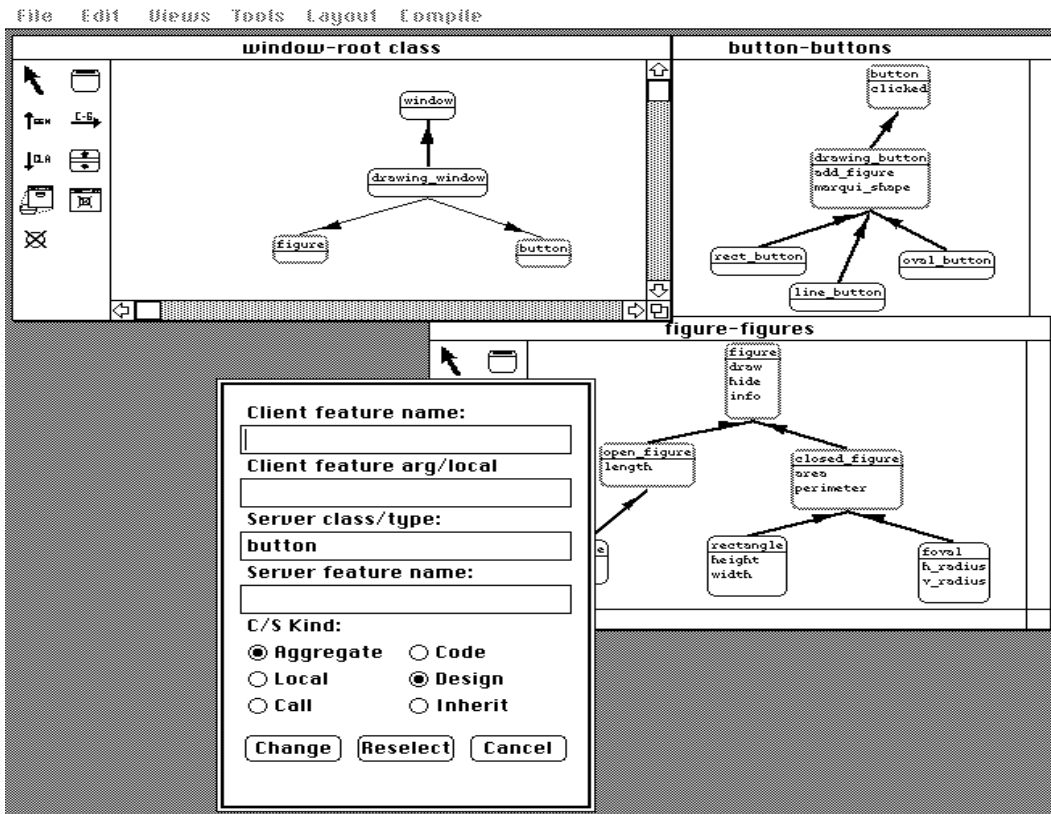
fig. 4.3.   Various features of classes and relationships between classes.

The next step is to define any other major aggregation and association relationships between classes and to define the main features of each class. Features[5] are added to class icons using the add feature name tool (⊞). No distinction between methods and attributes need be made during analysis. Fig. 4.3. shows the `root class` view after adding aggregation relationships between the `drawing_window`, `figure` and `button` classes (represented as thin arrowed lines). The dialogue shown is for entering information about an aggregation relationship. Note the type is "design" level which means the relationship is not directly related to any particular client-supplier implementation scheme. The other views in Fig. 4.3. have been refined from their Fig. 4.2. equivalents to show the names of important features which have been added to class icons. These features describe the important attributes (data) and methods (behaviour) associated with each class.

At the analysis stage documentation about the purpose of classes and relationships can be added using textual views. A textual view is defined using a menu item or the create view tool. Documentation describing the class, its features and its relationships to other classes can then be added. Documentation views can also be defined for individual features, as shown in fig. 4.4.

---

[5]Using the Eiffel terminology for all attributes and methods of a class (Meyer 88).

Currently SPE only supports this object-relationship modelling for OOA. Other useful techniques such as service charts and dataflow between objects (Fichman and Kemerer 92, Monarchi and Puhr 92) are not yet supported. Chapters 8 and 9 discuss extending SPE to support additional (and alternative) analysis and design diagrams.
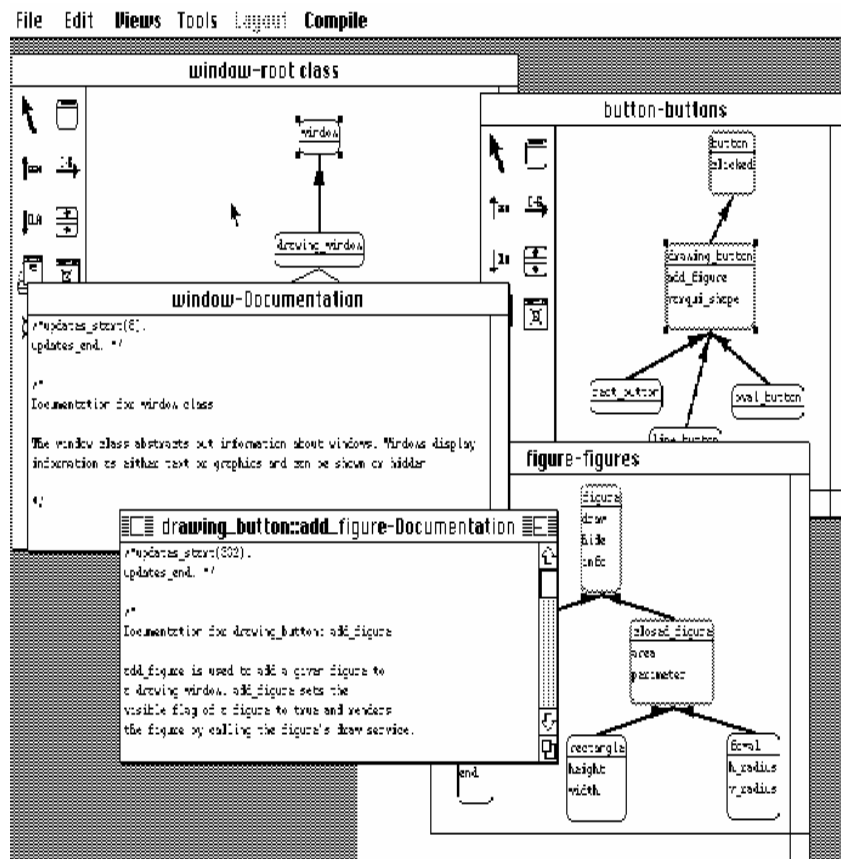
fig. 4.4.   Documentation for classes and features.

4.2.4. Design

After performing an analysis of the drawing program we can proceed to specify a design for its implementation. Extra detail is added to the various associations between classes, for example, the names to refer to them by and how they are to be implemented (as attributes, local or argument references, or by a feature call).

Fig. 4.5. shows further enhancement of the drawing program by adding design-level information to our analysis. Extra features and relationships between classes are introduced to implement various tasks. For example, to draw a figure a picture representing the figure's shape is added to a window and to delete it this picture is removed. The window classes must therefore support picture handling as an interface to an LPA Prolog graphics window. Note that the analysis-level diagrams and documentation can be retained or new views created by copying information and extending it. The documentation added at the analysis phase can also be extended here to

describe more detailed program structure. The connection points of lines on icons can be interactively modified to assist layout, as has been done in the "figure-drawing" view.
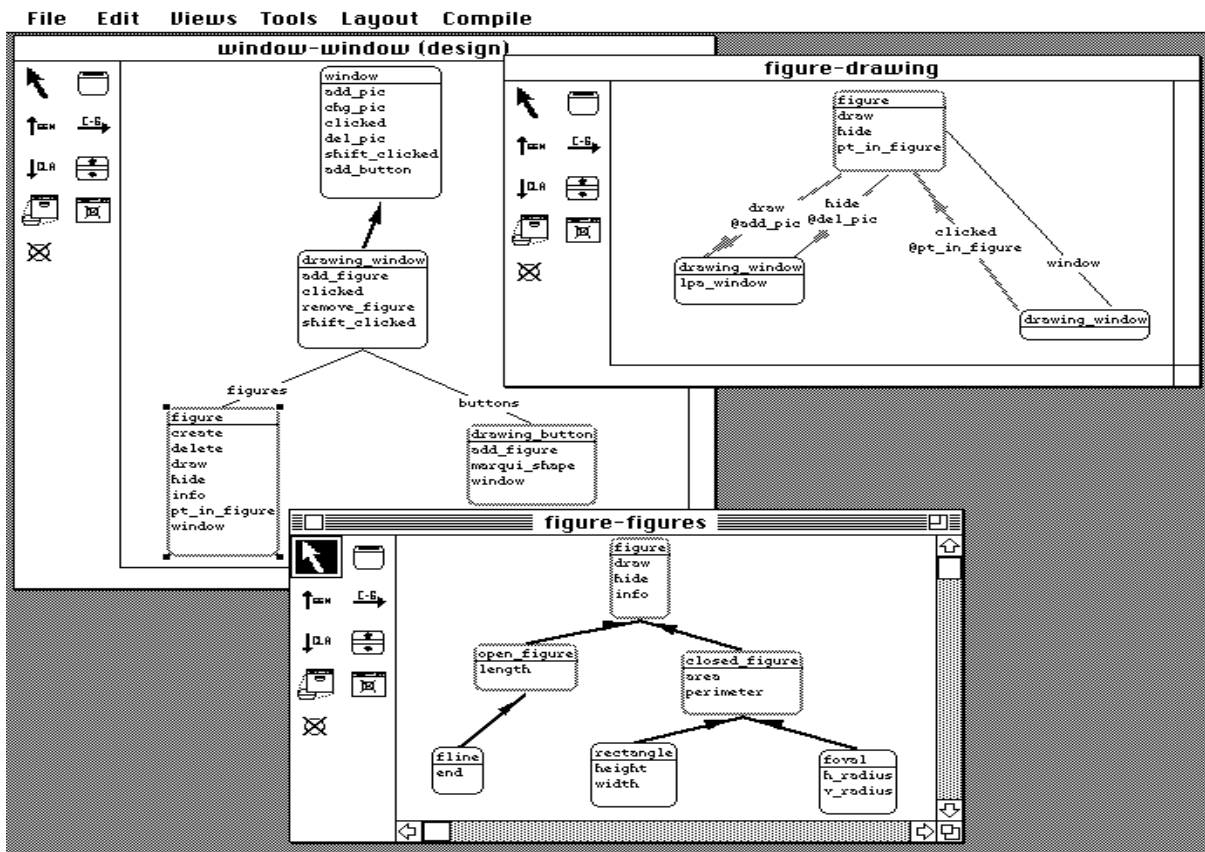
fig. 4.5.   Extending the analysis of the drawing program to a program design.

4.3.   Implementing a Snart Program

To implement the design in Snart the design diagrams can be extended to describe the actual types of client-supplier connections between classes. For example, the `figures` client-supplier relationship between the `drawing_window` and `figure` classes can be implemented as an attribute (aggregate) `figures` of type `list(figure)`. Other client-supplier connections, for example the `draw` feature of `figure` calling `add_pic` feature of `add_pic` are implemented as a method call of the form `Window@add_pic(Picture)`.

Class definition textual views are added to describe the complete set of features (including attribute types) and rename lists for each class. Textual views are created in a similar manner to graphical views, but consist of one or more "text forms"[6], rather than icons and glue. Textual views are manipulated by typing text in a normal manner. They also have an

---

[6]A text form is some text describing one aspect of a program component. A class can have documentation and code text forms, a method can have code and documentation forms, and an attribute only a documentation form.

alternative, high-level structure-oriented style of editing using menus for manipulating individual text forms. Methods are implemented in the same way as class definitions and can either be added to the same textual view as a class definition or have their own view (and window). Fig. 4.6. shows a class definition and methods implemented for the drawing program.
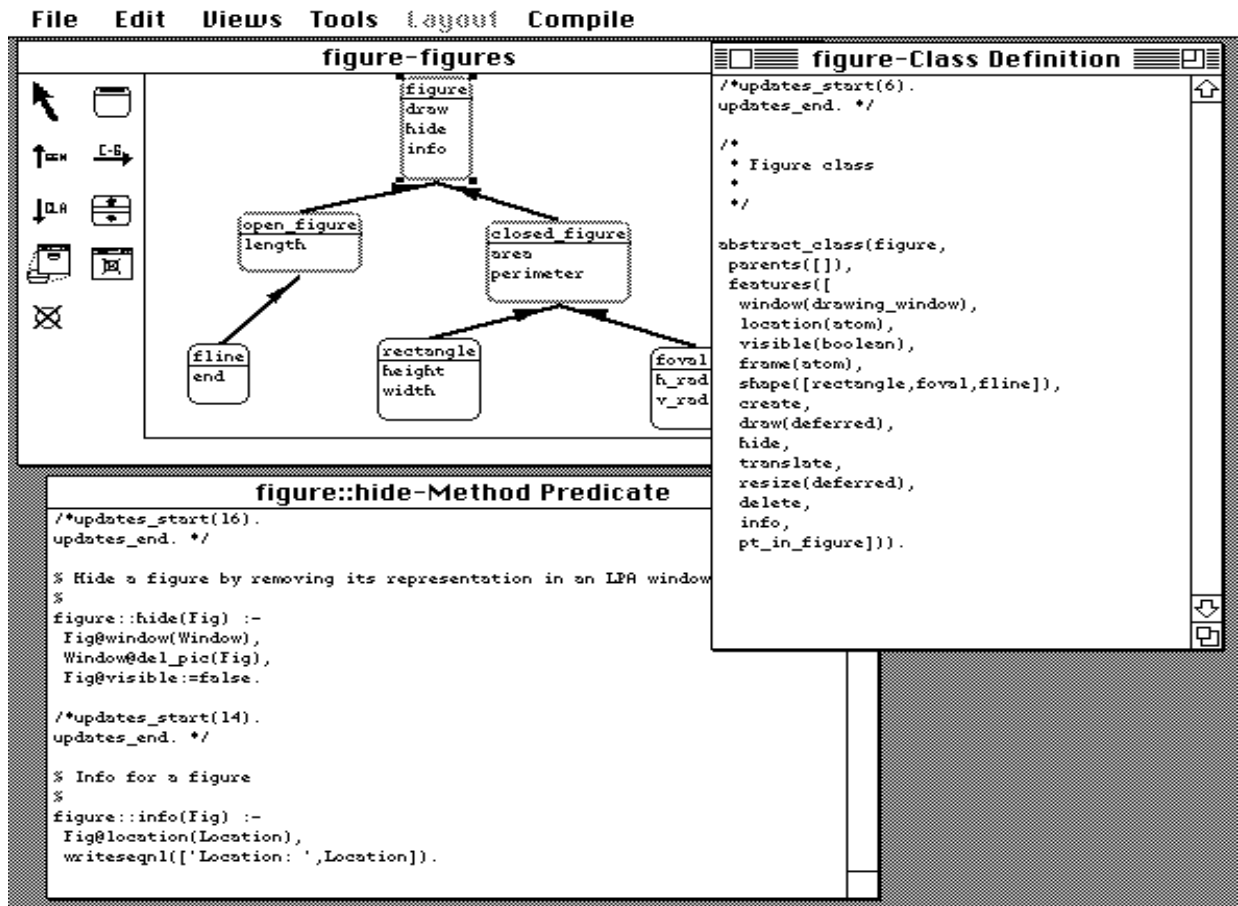
fig. 4.6.   Textual views implementing the drawing program from its design.

When a textual form is created, SPE generates a template from its design-level information. For example, when the class definition for `Snart` is created, the features `add_figure`, `marqui_shape` and `window` and parent class `button` are added to the class definition.

SPE textual views are parsed on programmer request. Snart code is generated by calling the existing Snart compiler with either a window or terms to compile. The Snart compiler regenerates the compiled definition of a class and its look-up tables when necessary. Any errors during parsing are reported using dialogues while compilation errors are reported by associating error messages with a program component (see Section 4.5).

4.4. Debugging a Snart Program

Once implemented, the program can be executed. Fig. 4.7. shows the drawing program running. The drawing program window is shown with some figures added via user interaction. Two "intra-object" debugging views are shown which display the state of drawing program objects. In addition, two program views show the `drawing_window` class definition and the drawing program hierarchy.
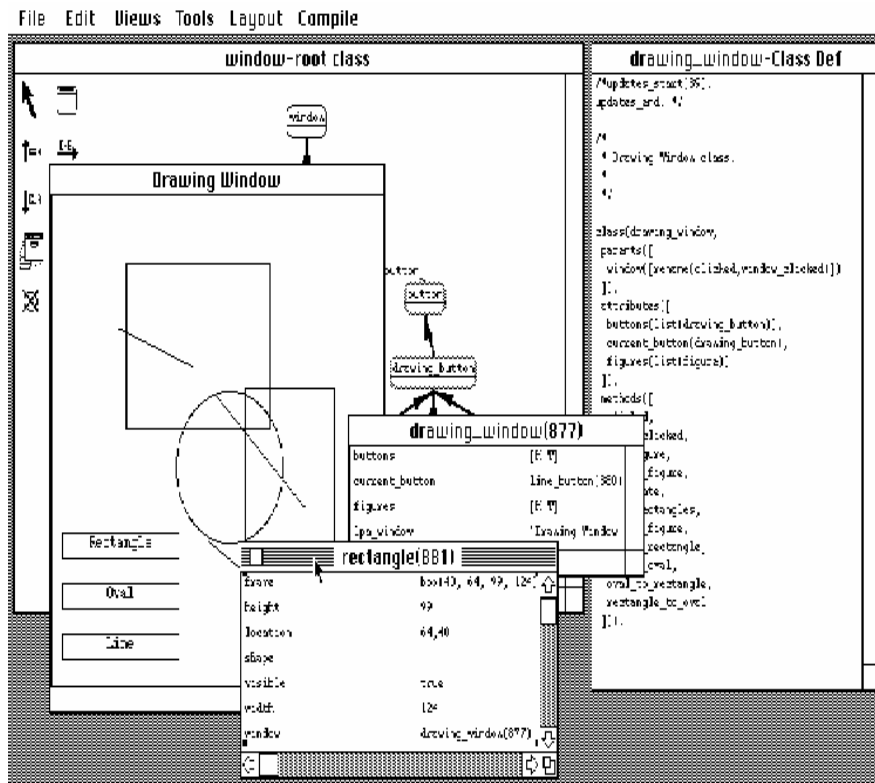
fig. 4.7.   The drawing program being debugged.

The SPE intra-object viewer shows all the attribute values associated with an object. Clicking on references to lists or other objects opens further object viewer windows. Errors in the drawing program can be fixed by modifying the program code in SPE. The LPA MacProlog debugger can be used to trace execution within predicates and methods in the normal way. Chapter 9 discusses some extensions to this simple debugging system for displaying object references and control-flow graphically.

4.5.   Modifying a Snart Program

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:

- The requirements for the program change impacting analysis, design, and implementation.

- A design may be incomplete and requires modification impacting on its implementation.
- Errors are discovered on execution, correction of which may result in design changes.

"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.

4.5.1. Graphical Updates

For graphical views, updates from textual manipulation and parsing or other graphical views are reflected by making the change directly to the icons in the view. If an aspect of a program has been deleted (for example, a feature moved to a sub-class), any inconsistent feature connection is drawn shaded or coloured to indicate the deletion.

Currently SPE only supports the propagation of updates on the same program component between different views. SPE often stores analysis and design relationships as separate components even if a design relationship is a more detailed representation of an analysis relationship. This is due to less information being provided at the analysis phase so relationships may not be uniquely identifiable. For example, Fig. 4.8. shows two views from the drawing program, one for analysis and one for design. The aggregation relationships in the analysis view are copied in the design view, but there is not always sufficient information to determine which analysis relationships correspond to which design ones.
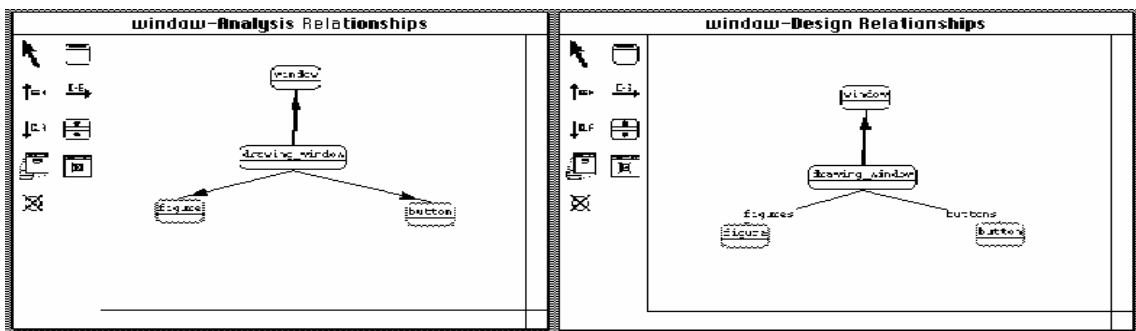
Fig. 4.8.   Analysis and design relationships.

An extension for providing analysis-to-design consistency (and vice-versa) where change to a generalised relationship affects more detailed versions of the relationship is proposed in Chapter 9. This supports the propagation of change between analysis and design relationships. It also describes how design views can be copied from analysis views and the two views kept consistent under change (i.e. the link between their relationships is inferred from the copying process itself).

4.5.2. Textual Updates

In textual views, changes are not immediately made to the rendered view. Rather, readable descriptions of any updates which have occured in other views are expanded into the view. The user then has an opportunity to accept, provide an implementation for, or reject each update. Both of the textual views in Fig. 4.9. include descriptions of updates which have been applied to other views. These update descriptions are expanded into any textual view, including documentation views.

```
drawing_window-Class Definition
/*updates_start(94).
update(32). % rename feature figures to gfigures
update(36). %  add client/server design call clicked :
-> figure : pt_in_figure
update(44). %  *** Compilation Error: Duplicate feature
names for clicked
updates_end. */

/*
 * Drawing Window class.
 *
 */

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

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

/*
 * Drawing Window class.
 *
 */

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

Fig. 4.9. (a) Updates expanded in a textual view, (b) first update applied.

The first update description in the `drawing_window` view indicates that the `gfigures` feature has had its name changed to `figures` in another view. A programmer can either:

- accept the change and have SPE modify the text appropriately (in this case, modifying `gfigures:list(figure)` to `figures:list(figure)`)
- implement the change manually or
- reject the change, causing the change to be undone throughout all views.

In some cases, such as the addition of a client-supplier relationship in a graphical view, it is not possible to automatically infer the correct modification to a textual view and user assistance is needed. For example, a client-supplier link 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.

Update descriptions may also be used to inform users of semantic or compilation errors (syntax errors are flagged interactively) and to document changes. For the latter, programmers can add arbitrary "user-defined" updates that describe various changes performed or to perform on a program. A compilation error is shown in the `drawing_window` Class Definition window in Fig. 4.9. and a user-defined update is shown in the same window in Fig. 4.10.

4.5.3. Update Histories

All the updates made to a program component may be viewed via a menu option, providing a persistent history of program modification. User-defined updates may also be added to document change at a high level of abstraction. Programmers may add extra textual documentation against individual updates to explain why the change was made and possibly who made it and when. Fig. 4.10. shows an example of viewing the updates for `drawing_window` and adding a user defined update for `drawing_window`. These updates describe each change that has been applied to the `drawing_window` class and are numbered in the sequence they were applied (updates 20, 21, and 22 in the list are visible in the update history browser dialogue). Update number 27 is having extra information added to more fully document the change in the update editor dialogue.
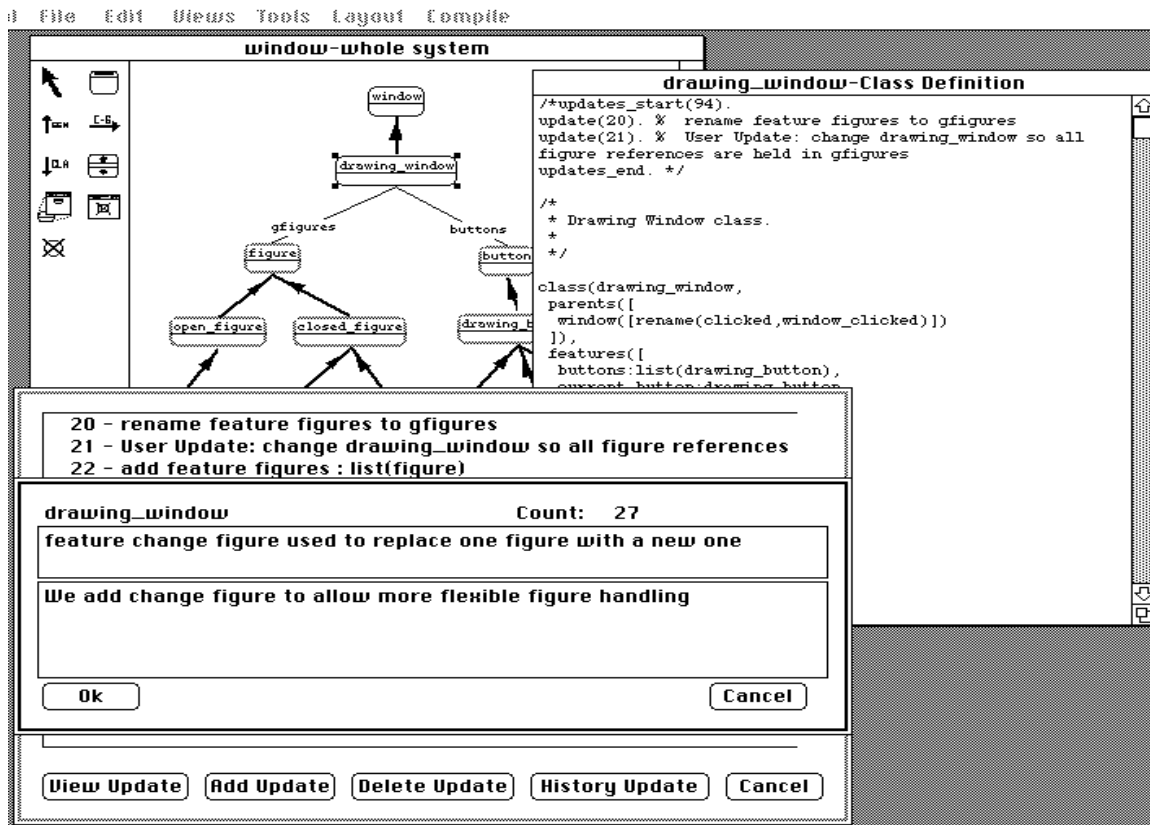
Fig. 4.10. The update history browser and update editor dialogues for `drawing_window` updates.

4.5.4. Integrated Software Development

Modifications to a program can be made at any level (analysis, design or implementation) and to any view. These modifications will be reflected in other views by direct update in graphical views or display of update descriptions in textual views. This produces a very integrated environment with little distinction being made between graphical or textual program manipulation. In fact, little explicit distinction is made between the different phases of software development, unlike other systems with different tools being employed for different phases of development (Wasserman and Pircher 1987). For example, if the drawing program requirements are extended so that wedge-shaped figures and arbitrary polygon figures are supported, these changes are made incrementally at each stage. Analysis views are extended to incorporate new figure and button classes, and new features are added to classes. Design-level views are extended to support the requirements of each new type of figure and implementation-level views are added or modified to implement these changes.

SPE propagates and stores the update descriptions that are displayed in textual views and the update history using *update records*. Update records, their generation, propagation and unparsing are described in Chapter 5.

4.6.   Browsing a Snart Program

SPE allows an arbitrary number of views to be created for any class or feature. Programmers must be able to locate information easily and be able to gain a high-level over-view of different program aspects.

Class icons in graphical views have "click-points" which allow programmers to double-click on the icon in a certain place and have some pre-determined action carried out (similar to Prograph's dataflow entities (Cox et al 89)). Fig. 4.11. shows the different click-points for a representative class icon.

Clicking on a *class views* point provides a list of views this class is a member of. Similarly, a *feature views* point provides a list of views a feature occurs in. A view can be selected from a views list dialogue and it will then become the new current view with its window brought to the front.

Class text points select a default textual view a class occurs in to become the current view. If the class does not yet have any text view, one will be created and will become both the current view and the default text view for the class. The kind of text form added to the view is determined by asking the programmer using a dialogue. Similarly, *feature text* points select or create the default text view for a feature.
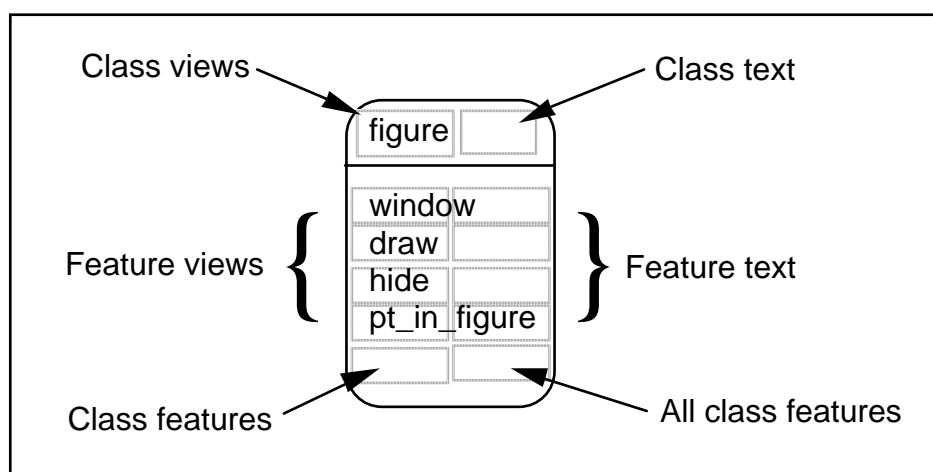
fig. 4.11. Click-points on a class icon to aid navigation.

Clicking on a *class features* point provides a list of all features defined for a class. Clicking on an *all class features* point provides a list of all features for the class, including inherited features. Any feature may be selected from these lists and its views or default text view made the current view. Option-clicking on a class features or all class features point provides a list of other class information including generalisation classes, specialisation classes, client-supplier relationships and classifiers. Feature names can be shown in or hidden from a class icon or a client-supplier relationship to model the feature expanded in the view. Other class information can be expanded on programmer request and SPE

automatically adds a graphical connection and class icon to represent the relationship(s). Fig. 4.12. shows the feature list and class information dialogues for SPE when browsing the features of the `figure` class.
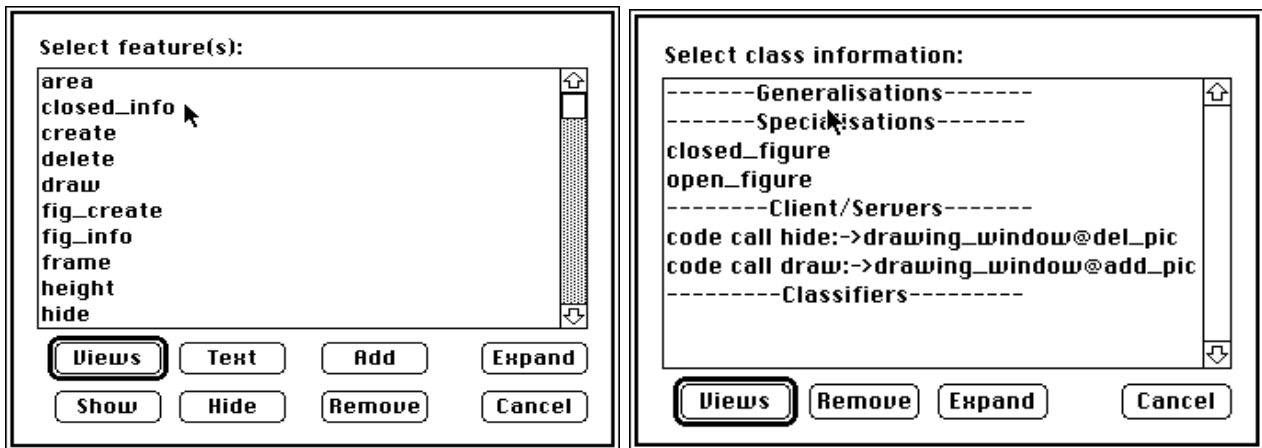
Fig. 4.12. The features selection and class information dialogues from SPE.

SPE provides menus for textual views that perform similar facilities to click points. Any elements whose text is in the selection range for a textual view (text highlighted by selecting with the mouse) can provide dialogues with feature lists or be manipulated like graphical icons (be hidden or their base component removed, their updates displayed in a dialogue, or their updates applied to the textual view).

SPE provides similar searching and find and replace facilities to LPA Prolog using the LPA search menu. A general location dialogue can be used to locate the views or default textual view for any class, feature or Prolog predicate. Fig. 4.13. shows an example using the general location dialogue.
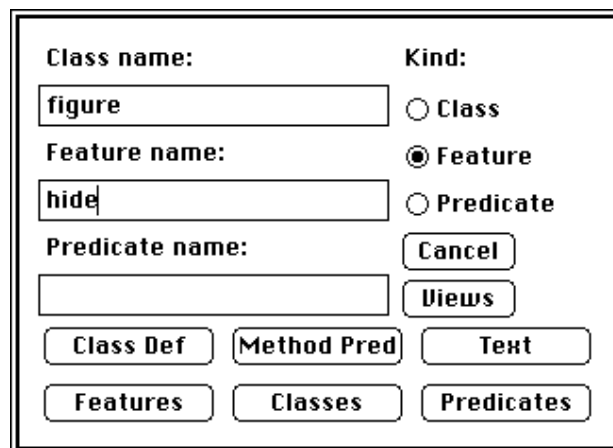
fig. 4.13. The general location dialogue from SPE.

Programmers can construct additional views for the sole purpose of program browsing. A graphical view can be constructed based on the selected icons in an old view. Already defined class information can be expanded into the view and used by programmers to

gain a different perspective on a program or just as a mechanism for accessing other views. This provides a very flexible static program visualisation mechanism with view composition and membership under the complete control of a programmer.

4.7. Managing a Snart Program's Complexity

When class inheritance hierarchies, such as those used in the drawing program, are constructed software becomes much more complex. Information that comprises the full interface for a class can be stored in many ancestor classes. In addition, client-supplier relationships between classes mean control flow travels through many different methods associated with different classes. SPE provides various complexity management facilities which allow programmers to define extra views to reduce the cognitive complexity of programs. Complexity management is a similar yet distinct concept from program browsing and can greatly affect the usefulness of any browsing strategies.

4.7.1. Programmer-defined View and Icon Composition

Programmers determine which features are represented in class icons. A view might contain information focusing on one particular aspect of a class or show only those features of a class relevant to other classes in the view. Fig. 4.14. illustrates a view from the drawing program showing the relationships between the figure and `drawing_window` classes. Only the relevant features of each class are shown.
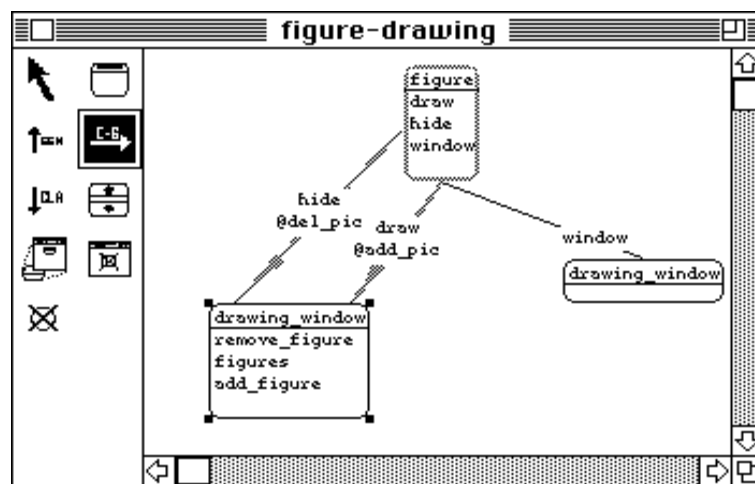
Fig. 4.14. A view showing client-supplier relationships between `figure` and `drawing_window`.

Textual views can contain several text forms for different program components which is useful for representing strongly related or inter-dependent program aspects together. For example, the `figure::hide`, `drawing_window::del_pic` and `drawing_window::remove_figure` methods are all related by client-supplier relationships. All three methods can be represented in a single textual view and will thus always be displayed together when the view is selected.

4.7.2. Code Text Forms

Class text forms can be either canonical, documentation or a "code view". Canonical class definitions show the features defined by a class and all generalisation information used by the class (classes it inherits from and features it renames). Canonical forms are given to the Snart compiler to regenerate a class's dispatch tables and class information. Class documentation describes any additional user-defined information about a class and there may be more than one documentation form for a class (describing different, related aspects of the class).

Code view text forms have a similar syntax to canonical class definitions but can represent inherited features and a subset of a class's features. For example, the three methods for hiding figures could be displayed with two code view text forms of the `drawing_window` and `figure` classes which only represent the relevant class features. Fig. 4.15. shows the contents of a textual view containing these five text forms. Note that updates on one text form for a class will be propagated to views containing any other text form.

A class or feature can have several documentation text forms which can be shown together or in any combination with code forms. SPE doesn't directly support a notion of responsibilities for classes or feature membership of responsibilities (Wirfs-Brock and Wilkerson 89). We can model such a system, however, by using code views of classes with only features belonging to one or more responsibilities being shown.

```
/*updates_start(16).                          /*updates_start(10).
updates_end. */                               updates_end. */

% Hide a figure by removing its             class(drawing_window,
representation                                  parents([
% in an LPA window.
%                                             window([rename(clicked,window_clicked)])
figure::hide(Fig) :-                            ]),
  Fig@window(Window),                           features([
  Window@del_pic(Fig),                            remove_figure,
  Fig@visible:=false.                             inherited del_pic
                                                ])).
/*updates_start(81).
updates_end. */                               /*updates_start(308).
                                              updates_end. */
% Delete a picture from this window
%                                             abstract_class(figure,
window::del_pic(Window,Name) :-                 parents([]),
  Window@lpa_window(LPA),                       features([
  Window@make_name(Name,PictureName),             delete(method),
  del_pic(LPA,PictureName).                       hide(method),
                                                  window:drawing_window
/*updates_start(102).                           ])).
updates_end. */

% Remove a figure from the figure list of
this window.
%
drawing_window::remove_figure(Window,
    Figure) :-
  Figure@info,
  Window@figures(Figures),
  remove(Figure,Figures,NewFigures),
  Window@figures:=NewFigures.
```

4.7.3. Inheritance

Classes inherit much of their information from their ancestors. Allowing provision for inherited features and relationships in a view enhances the descriptive power of SPE's class views. SPE allows programmers to expand inherited relationships or designate relationships as "inherited". SPE can also check for changes to inherited relationships and class icon feature names and thus update them after change or inform programmers of inconsistencies.

SPE's browsing capabilities and complexity management mutually complement each other. Feature dialogues allow both class-owned and inherited features to be expanded in a view. A class or feature can be located by browsing and then added to another textual or graphical view. Reducing the number of features and class relationships in a view makes browsing and understanding of program sub-components much easier than if complete class details are always provided.

4.8. Saving and Reloading a Snart Program

Snart programs are saved and loaded as projects which contain all information about a program. Snart incrementally saves projects so only updated information is re-written to project files. Views and class data are incrementally loaded when required. Only a subset of a program's entire set of views and class data is held in memory at one time. Upon re-opening a project all views visible when the project was saved are reloaded and redisplayed, and the class, feature and predicate sets for a program rebuilt. Any further views are reloaded from the project file when a programmer selects them for display. Class data is loaded when accessed (when the class is displayed in a view or used in a selection dialogue).

If a reloaded view contains information that is out-of-date (i.e. the program has been changed while the view was not in memory to be updated) SPE indicates inconsistent information to the programmer. Textual views have updates expanded, and any textual forms in the view no longer consistent with the program data (for example, a method that has been deleted, renamed or moved to a sub-class) are rendered as "unmapped". For graphical view components with out-of-date information, the icon is rendered in a different colour, for example as "needs updating" (green) or "unmapped" (red). Programmers can then decide on the appropriate action to take to make the view consistent with the new program state: change graphical or textual component data, apply updates, or delete the view component.

4.9.   Discussion and Possible Extensions to SPE

In this section we evaluate SPE and determine possible future extensions to the environment. Chapters 5, 7 and 8 discuss how some of these enhancements could be modelled and implemented.

4.9.1.   Adequacy of Program Representation

The current representative power of SPE is good for describing the state of classes and their inter-relationships with other classes. SPE supports analysis, design and implementation-level descriptions with successive refinement of detail. Multiple views allow class diagrams to contain only information a programmer deems relevant for a particular focus. Documentation and code views provide a flexible mechanism for allowing users to add detail about different aspects of a program.

SPE does not provide any specific diagrams for modelling class behaviour (at the analysis or design levels). Client-supplier relationships can be used to model feature calls between classes but these can not be as specific as Service Charts (Coad and Yourdon 91) or Action-dataflow diagrams (Fichman and Kemerer 92). Some form of object lifecycle diagram and/or dataflow diagram for modelling the detailed interactions between objects would be very useful for analysis and design refinement. Chapters 7 and 9 propose examples of such diagrams for SPE.

Abstract class relationships such as aggregation and association (Henderson-Sellers and Edwards 90, Coad and Yourdon 91) are currently modelled as "abstract client-supplier" relationships which is not always the most convenient or descriptive approach. Dialogues should support more context-dependent interaction including referring to client-supplier relationships as "aggregation and association" for analysis (Henderson-Sellers and Edwards 90), and showing either high-level feature information or feature detail depending on whether a view is "design-level" or "code-level". Class contract views that support Eiffel-like pre- and post-conditions and invariants (Meyer 92) and additional documentation would provide a more abstract and expressive specification than selective views of class code with documentation.

SPE does not currently support any notion of responsibilities (Wirfs-Brock and Wilkerson 89). These would be useful for grouping related features so they can be viewed as a group with the same responsibility in both graphical and textual views. Filtering mechanisms are not currently provided for features with programmers determining which features are shown in a class icon, class definition views and feature selection dialogues. Using filters in conjunction with responsibilities would allow SPE to support a notion of partial views of a class for certain responsibilities (for example, the hiding of figures example from Section 4.7).

It would be useful if programmers could mark features as "over-rideable", "must over-ride", "must not over-ride" and so on under inheritance. This would allow SPE to check whether features have been correctly re-used in sub-classes and would also allow SPE to generate more complete templates and views for classes (possibly including the argument names and types for over-ridden methods). Documentation of views and program elements for reuse extends this concept of SPE-supported reusability even further.

4.9.2. Program Viewing and Construction Facilities

SPE provides basic class diagram construction facilities with some flexibility provided via programmer-defined view composition and layout (programmers determine which classes and features are shown and their positions). Class relationship connections (generalisations, client-suppliers and so on) can be attached to different parts of a class to allow different diagram layouts and multiple connections to and from one class icon.

Programmers are not given much control over icon shape and composition, menu options, tools, preferences and other SPE facilities. Interaction via menus and dialogues is fixed with no ability to define macro editing operations or define commonly used default values for dialogues. Allowing more flexible diagram composition, as in TurboCASE (StructSoft 92), would give programmers a more comprehensive diagramming capability. Icon and connector shapes, sizes and layout are currently determined by SPE. Programmers should be able to move individual icon components to suit their requirements and be able to "bend" lines at appropriate points to aid diagram layout and readability.

SPE does not support automatic layout of diagram components except when expanding relationships and classes from a class icon. Lines can only be direct connections without multi-point lines. Features are shown as a group with no ability to move them, connect lines directly to them to represent feature-to-feature connections (as supported by some CASE tools, such as TurboCASE (StructSoft 92)), or show arguments for method calls. Resizing of icons to reflect importance or to change a view layout is not provided. Modification of program data is mostly by dialogues when it would sometimes be useful to allow direct manipulation of icons or icon components (for example, renaming a feature connection by editing its name rather than via a dialogue). A possible inter-feature relationship view is shown in fig. 4.16.
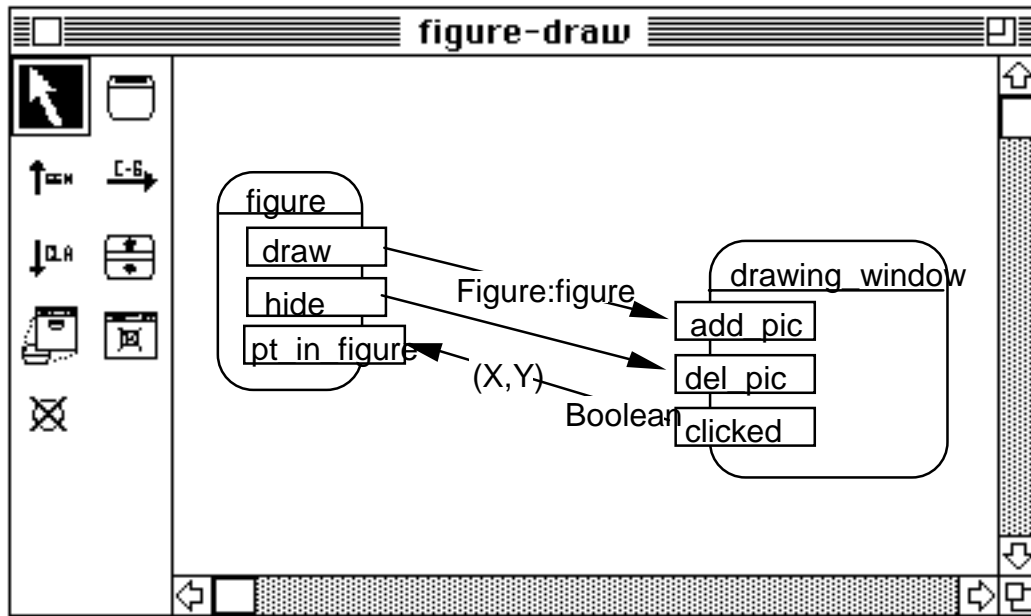
fig. 4.16. A possible form for SPE inter-feature relationship views.

The text editor provided by SPE is a standard Macintosh text editor with various menus provided to manipulate textual forms and move to other views. While this provides a well-integrated editing environment, many improvements could be made to enhance productivity. Better separation of textual forms, updates in pop-up menus, hidden textual form boundaries, structure-oriented editing of class features, and hyper-text macro commands would all be useful. This would require an editor with the structure-oriented/free-edit capabilities of UQ2 (Welsh et al 91), annotation capabilities similar to FIELD (Reiss, 90b), and user-defined unparsing as supported by MELD (Garlan 86) and Mjølner (Minör 91). Allowing textual and graphical representations to be integrated within one window, similar to Dora (Ratcliffe et al 92), may also be useful.

Debugging is integrated with conventional Prolog debugging by using the LPA debugger and an object browser provided by SPE. The object browser provides much improved access to Snart objects than the original Snart environment's object printer. Higher-level debugging facilities would be useful, however, including a graphical representation of object references (possibly a subset of an object's references) similar to class diagram views and Cerno's inter-object views (Fenwick 93).

4.9.3. Large-scale Program Development

Update descriptions provide a good way of maintaining consistency between graphical and textual views, automatically documenting program changes, reporting compile-time and semantic errors, and allowing users to add their own updates or browse the "update history" of part of a program. Change propagation between design and implementation is supported but not between design and analysis. Updates could provide a mechanism for propagating changes to sub-classes and associated classes when a class is modified. A

browsing system for updates is provided but this could be extended to allow "classes with errors" and "classes with unseen updates" to be located and modified (i.e. a grass-catcher similar to that provided by the Trellis/Owl environment (O'Brien et al 87)).

SPE does not currently support version control (multiple forms of a program to exist), shared libraries (a program made up of one or more shared components), or multi-user development (one program being constructed by more than one user on several machines, with concurrent access and updates to a program, similar to (Nascimento and Dollimore, 93)). Extending the environment to allow multiple access (possibly con-current) to a program would allow much larger systems to be implemented as a group effort. See Chapter 10 for further discussion on extending SPE for multi-user software development.

4.9.4. Programming Other Languages with SPE

The facilities provided by SPE are quite general and applicable to a wide range of object-oriented languages. Generalising the environment and supporting other languages such as Kea, Eiffel and C++ in a similar manner to Snart would provide development environments of similar capability to SPE. Adding typed variables to Snart (as discussed in Section 3.8) would allow both compile-time checking of errors in SPE and detection of concrete client-supplier relationships. These could be used to generate call graphs or provide checking for abstract relationships defined in SPE itself. Using any strongly-typed object-oriented language (such as Eiffel) would also allow compile-time generation of such information.

4.10. Summary

The simple programming environment for Snart under LPA MacProlog provides only basic support for object-oriented programming. No analysis or design tools are provided or interfaced to and there is no support for automating program documentation and version control. The LPA debugger is used to examine executing predicates with very simple facilities to print run-time object data.

SPE provides a sophisticated development environment for Snart. SPE supports analysis, design, implementation and maintenance of Snart programs in one environment. Unlike most CASE tools, changes at one level are propagated to other levels and graphical and textual representations kept consistent. Snart programs can be run and debugged in the same environment and extra views created for browsing and complexity management. A novel form of human-readable update descriptions are used for view consistency, documentation of change, and semantic and compile-time error reporting. SPE can be improved in various ways but provides the basis for a complete, integrated development environment for Snart.

The basic features of SPE can be factored out into two distinct systems. The concepts of multiple view support with consistency management are reusable for other programming environments. Chapter 5 presents a model for describing such environments and two languages for specifying such systems: one for describing the state of a program and how it can be manipulated, the other for describing user interactions for viewing and changing a program. Chapter 6 describes an object-oriented architecture for such environments based on the model of Chapter 5 and Chapter 7 presents an object-oriented implementation based on this architecture. Chapter 8 shows how SPE can be generalised into a generic programming environment for object-oriented languages. It also discusses how this generic environment can be modelled and implemented using the architecture and implementation framework from Chapters 6 and 7 respectively.

Chapter 5

Modelling and Specifying Environments with MViews

As discussed in Chapter 2, the multiple view aspects of software development environments, such as SPE, are common to many different environments. It would greatly simplify the construction of such environments if a common set of building blocks was available. This should support multiple textual and graphical views, automatic view consistency management, a flexible program representation, and provide support for user interface construction and program and view persistency.

MViews abstracts out the common features of environments that support multiple textual and graphical views of information. Programs are represented as object dependency graphs, subsets of these graphs are constructed to form multiple views of a program, views can contain many instances of different program elements, and views can be rendered and manipulated in either a graphical or textual form. The rationale for MViews is discussed together with the deficiencies with current approaches to providing environment construction facilities. A model for MViews-based environments is presented and a declarative specification language developed. This language captures the fundamental concepts of the environment and provides a mechanism for abstractly specifying program and view state and the semantics of manipulating this state. An operational semantics specification of this language is presented to show that the language is well-defined in terms of the MViews model. A complementary visual specification language is introduced for defining the appearance and interaction with MViews environments using a graphical style. This visual specification provides the input and output mechanism for the state description.

Chapter 6 further develops the MViews model by providing an object-oriented architecture for constructing such environments based on the model and specification languages. Classes from this architecture are specialised to define environment-specific program and view representations. Chapter 7 describes an object-oriented implementation of MViews as a framework of Snart classes based on the architecture of Chapter 6. Chapter 8 illustrates the use of this architecture and framework by describing an architecture and implementation for SPE.

5.1. Rationale For MViews

Section 4.1 discussed some desired features of software development environments. These include integrated analysis, design and implementation of software (including using the same representations during different phases of software development); graphical and textual views of programs; multiple views of software including consistency management and recording of change; and tool integration with a common user interface and data management strategy. A model for such environments should thus provide several key aspects which are discussed below.

5.1.1. Program Representation

Program structure needs to be represented in a manner which is both flexible (so many different application structures can be represented) and close to the application's needs (i.e. a "natural" representation for the application domain) (Meyers 91, Arefi et al 90). It should also be generalised sufficiently so reuse of the model is possible but does not involve great effort (i.e. be a sufficiently abstract modelling of program structures) (Minör 90, Backlund et al 90). A language representation scheme also requires a mechanism for describing language-specific semantics and this scheme should be complementary to the program structure representation (possibly using structural modifications to drive semantic checking) (Reps and Teitelbaum 87, Backlund et al 90, Minör 90).

5.1.2. Multiple Textual and Graphical Views

Environments like SPE and Dora require both graphical and textual representations of parts of a program (Grundy and Hosking 93, Ratcliffe et al 92). Thus views should support a model of a subset of the total program state and also provide either a textual or graphical rendering of this "partial program". The structure of views should be similar or the same as the structure of base program data. This allows view data to be manipulated in a similar manner to the base program it mirrors and consistent manipulation strategies to be employed throughout the environment (Vlissides 90). There should, however, be some scope for structuring views differently for efficiency or because a different structure is a more appropriate model for the view (Dannenburg 91).

5.1.3. Program and View Modification

Program structures must be modified to construct or change program fragments. Similarly, modification of a view is equivalent to a programmer changing the part of a program displayed in the view (Meyers 91, Vlissides 90). View editing operations should thus be translated into appropriate program modifications. Using the operation model for a base program for changing a view's state (as opposed to its rendering) may be appropriate to help facilitate this translation process.

Programmers generally find free-editing text and interactively editing graphics is a more natural and desired approach to programming than comparable structure-editing approaches (Welsh et al 91, Arefi et al 90, Minör 90, Whittle et al 92). Environments should thus support editing mechanisms that match those desired by programmers and which enhance their productivity, rather than using particular editing mechanisms because they are the easiest to support with environment generators. An editor which provides free-editing and incremental parsing for syntax and semantics checking, and can support structure-editing in a consistent manner for high-level programming, may be the best solution (Welsh et al 92).

5.1.4. Automatic, Efficient Consistency Management

When a program component is updated all affected views should also be updated to reflect the change (Meyers 91, Reiss 85). The change should also cause any language-specific semantics to be rechecked to ensure programmers are informed of errors (Reps and Teitelbaum 87, Minör 90). This change propagation process should be both efficient and as automatic as possible so programmers need not be concerned with inter-component dependencies (Reiss 86). Lazy application of updates may be appropriate for view updates when the view is hidden or not in the front editing window (Dannenburg 91, Wilk 91). Attribute recalculation for semantic checking (Reps and Teitelbaum 87), where affected values need not be recalculated until they are required (Hudson 90). Incremental view updating, where only the updated aspects of views are re-rendered rather than the whole view for efficiency, should be directly supported (Vlissides 90, Dannenburg 91). It would be useful to incrementally update a view given changes to its base rather than have to compute the changes (Dannenburg 91, Wilk 91). Views should also be visually updated to indicate that a change has occured in the program state that may not necessarily be possible for the environment to *automatically* translate into appropriate view modifications. For example, SPE does this when propagating a client-supplier addition or deletion to a textual class code view.

5.1.5. Recording Previous Changes

The modification history of a program component can be useful to inform programmers of what changes were made to the component, when they were made (relative to other changes), and possibly who made the change and why. Providing on-line access to this change history would allow it to be used as a documentation aid.

5.1.6. Undo and redo of User Manipulations

Editors on views should support some form of undo/redo facility to allow programmers to reverse editing operations that may have had the wrong or unintended effect (Reiss 85, Vlissides 90, Dannenburg 91). This undo/redo mechanism should be abstract enough so

programmers do not need to be concerned with implementing such a facility directly and be efficient enough so the mechanism does not impose unacceptable storage or performance demands on the environment (Dannenburg 91).

5.1.7. Program and View Persistency and Multi-user Access

Programs and their views need to be stored between invocations of an environment. The environment may also need to support multi-user access to a program (possibly with distributed copies of the program and multiple versions) (Meyers 91). Ideally, program persistency should be efficient in both time and space, require little or no application-specific programming to support, and be flexible enough for the different requirements of different environments.

5.1.8. Tool Integration and Extensibility

Environments are typically made up of several tools used for different purposes, for example editing, compiling, debugging and version control. Environment integration should be at both the user interface level (providing a consistent user interface across all tools) and the tool data level (providing uniform data storage or translation mechanisms) (Meyers 91, Wang et al 92, Reiss 90a). An environment should also be extensible, allowing new tools to be developed or existing tools from other systems to be integrated in a consistent manner (Meyers 91, Reiss 90a, Wasserman and Pircher 87).

5.2. Related Research

This section discusses related research on viewing mechanisms for programming environments and related applications. It illustrates that most existing systems, while supporting some of the requirements of Section 5.1., do not provide enough support for all of these requirements.

5.2.1. Smalltalk Model-View-Controller

The Smalltalk Model-View-Controller (MVC) model (Goldberg and Robson 84) provides a general mechanism for representing base programs (a model) as Smalltalk objects but with no specific support for programming language structure or semantics representation. Views of model objects can be defined which are objects linked to the model objects they view. Models are updated by object manipulation while changes to views are translated into model changes by window-based editors (controllers).

Views are notified of changes to their model objects by a simple "update yourself" mechanism. View objects are sent an `Update` message which indicates their model has changed in some way and they must reconcile their state to that of their model's. As explicit model changes are not sent to view objects, it is often difficult (or impossible) to

determine the exact model change that occurred. Hence a view may be required to do more work that strictly necessary to reconcile its state to its model's state (Wilk 91, Dannenburg 91). For example, if a model (base) component has an item added to a list attribute, affected views may not be able to determine this exact change and hence may need to totally redisplay themselves. MVC does not provide any specific mechanisms for incremental view updating, lazy updating, visually indicating model changes that can't be explicitly applied to a view, change history recording, undoing or redoing view edits, or program persistency.

5.2.2. Interviews and Unidraw

Interviews (Linton et al 89) provides a framework for constructing graphical user interfaces. Unidraw (Vlissides 90) provides a framework for constructing domain-specific graphical editors. The Unidraw model assumes "programs" are hierarchically structured graphical objects with attributes. Changes to attributes can be propagated using a dataflow mechanism where changes to state variables are sent to dependent variables. A subject-view metaphor (similar to MVC) is used to support multiple views of a base data structure and subjects and views use a common structure and command scheme. Unidraw assumes a graphical representation and editing mechanism for views with no direct support for textual representations. Data is modified using commands (editing operations) and view updates are translated into program updates using an editor (similar to an MVC controller) and editor tools.

The view updating scheme is similar to MVC with a `Notify`/`Update` model, but Unidraw supplies a "damage" algorithm which automatically reconciles a view's state to its subject's. The disadvantage with this approach is that Unidraw assumes views have exactly the same structure as their subject (though this can apparently be changed via sub-classing (Vlissides 90)). View to base updates are handled in an application-specific fashion by the view's editing tools and manipulators.

Commands provide an undo/redo facility for editing operations but no change recording mechanism is supported (though it may be possible to build one by recording command objects). A simple database-like component persistency model is supported which allows subject and view structures to be written to and reloaded from persistent storage in an application-specific manner. There is no support for integrating existing tools except for a simple data export facility (typically to a textual form, for example Postscript).

5.2.3. PECAN, GARDEN and FIELD

PECAN (Reiss 85) provides an integrated environment for Pascal programming using multiple textual and graphical representations of a common program. PECAN provides a program representation and semantics calculation model based on trees. Multiple

graphical and textual views are supported but graphical views use a structure-edited approach while textual views use an incremental parsing algorithm with limited editing flexibility. View updates are via a MVC-like model and views provide translation mechanisms to map editing changes to base program changes. PECAN does not support update recording or version control but does have a flexible undo/redo facility which includes macro-operations. Programs are stored in files and there is apparently no support for incremental program persistency. Kaiser notes that the PECAN model would be difficult for most people to reuse due to its complexity (Kaiser 85).

GARDEN (Reiss 86 and 87) provides an environment for prototyping visual programming languages and for conceptual programming with several different languages. All data is represented by objects which provide a structural (syntactic) language representation scheme and also provide support for both static and dynamic language semantics. Views are defined as dependencies between objects moderated by a third object. View updates are translated to and from base program updates using this dependency model with changes being indicated in a MVC-like manner (i.e. an "update yourself" message is sent to the moderator of the dependency which then propagates the change). GARDEN uses an object-oriented database for program storage and to implement an undo/redo scheme (using transactions). While this is very general, Reiss notes it can have performance problems and difficulties in providing for environment evolution and existing tool integration (Reiss 86, Reiss 90b)

FIELD environments (Reiss 90a and 90b) provide the appearance of an integrated programming environment built on top of distinct Unix tools. Program representation is usually as text files with each tool supporting its own semantics (currently with a conventional compiler and debugger). Views are not directly supported but tool communication via selective broadcasting (Reiss 90a) allows changes in one tool "view" (for example, an editor) to be sent to another tool "view" (for example, the debugger). Free-edited textual program views are supported (but these text views cannot contain over-lapping information) while graphical representations are generated from cross-reference information. Reiss notes that a lack of user-defined layout and view composition for these graphical views is a problem (Reiss 90b). Version control is not currently supported and undo/redo is left to appropriate tools to support. Persistency is via Unix text files and a simple relational database (for cross-reference information). New and existing tool integration (and hence environment extensibility) is supported by providing a user interface (constructed from standard building-blocks) and selective broadcast entries for these tools.

5.2.4. Grammar-based Environment Generators

Chapter 2 briefly discussed several grammar-based programming environment generators including The Synthesizer Generator (Reps and Teitelbaum 87), Mjølner/ORM (Minör 90),

LOGGIE (Backlund et al 90), Dora (Ratcliffe et al 92), and MELD (Kaiser and Garlan 88). Most generated environments provide very abstract structure and semantics specifications (Whittle et al 92). However, most environments generated by these tools use a style of editing not yet well accepted by programmers (Minör 90, Welsh et al 91). Attribute grammars do not always provide an efficient means of recomputing semantic values as values must usually be recomputed entirely no matter what the change in values they use (Wilk 91). In addition, they suffer from not being fully-fledged programming languages and thus can lack power of expression for various tasks (Kaiser 85). As such systems are based on specific models for environments, modelling different interaction mechanisms or structures using these tools is not usually possible. Multiple textual and graphical support in such environments is often rudimentary or not directly provided, as are undo/redo, flexible program persistency and program change documentation. Tool integration mechanisms are provided by Dora (via a PCTE database repository (Wang et al 92)), MELD (static tool views (Garlan 86)) and Mjølner (using a backbone structure based on Unix files (Minör 90)). These use view-based and file system integration, however, which usually makes new tool integration and extisting tool extensibility difficult (Meyers 91).

5.2.5. Dannenburg's ItemList Structure

Dannenburg's `ItemList` structure (Dannenburg 91) represents data as a list of `Items` which have multiple, tagged values to support versioning. This representation scheme is cumbersome for representing programs (the only data structure directly supportable is the list) and provides no language semantics support. The `ItemList` supports multiple views which are themselves `ItemLists` and are updated by indicating which `Items` have been changed at the base level. A flexible undo/redo facility, incremental view updates, and automated base-to-view and view-to-base update propagation is supported. While the `ItemList` records old updates, these are stored against each `Item` value with no application-level access and hence couldn't be used to document changes to a program component. The `ItemList` does not directly provide a persistency model or support for describing updates that can't be directly applied to an `ItemList` view.

5.2.6. Wilk's Object Dependency Graphs

(Wilk 91) describes an object dependency graph (ODG) representation which stores data as objects with a dependency relationship network over the objects. ODG provides no language semantics scheme and no direct support for multiple views, though both could be modelled using object dependency. Updates to an object are propagated using change reports which describe the exact change that has occurred to an object and its dependency relationships (including changes to the objects' components via part-of relationships). Dependent objects can make exact changes to their state based on the change reports of objects their state depends on. Lazy consistency management is supported and transient update propagation also provided. This system does not provide undo/redo of changes

nor does it provide a program persistency mechanism. While change reports can be used to achieve consistency they can not be stored long-term to document program component changes.

5.2.7. Summary

Table 5.1. summarises the facilities provided by different multiple view and programming environment construction systems discussed previously. It also illustrates how the MViews model for programming environments described in Section 5.3. satisfies these requirements.

	MVC	Unidraw	GARDEN	FIELD	LOGGIE	Mjølner	Dora	ItemList	ODG	MViews
Program/Data Representation	objects	objects	objects	text (files)	abstract syntax trees with garlands	abstract syntax trees	objects	ItemLists	objects with object dependencies	object dependency graphs
Mutliple Views	model-view	subject-view	object dependency	partially via selective broad-casting	model-view	not supported	subject-view	Item dependency	not directly supported	object dependency graphs
Program and View Modifications	appl-specific	graphics interactive tools	text free-edited, graphics structure-edited	text free-edited	graphics structure-edited	text structure-edited	text and graphics structure-edited	appl.-specific	appl.-specific	text free-edited, graphics interactive tools
Consistency management	Update message sent to views	Update message sent to views	via object dependency	selective broad-casting between tools	Update messages, attribute grammar	attribute grammar	Update message sent to views	Items marked as Updated	change reports	update records
Incremental View Updates	no	damage algorithm	not directly	no	no	no	damage algorithm	Item versions	no	update records
Change Recording	no	no	no	version control	no	version control	no	no	no	update records
Undo/Redo	controller-specific	command objects	database transactions	editor-specific	editor-specific	editor-specific	editor-specific	Item versions	no	update records
Persistency	appl.-specific	catalogue (as text files)	database	text files	appl.-specific	backbone (as text files)	PCTE database	appl.-specific	appl.-specific	appl.-specific
Tool Integration and Extensibility	no	no	no	selective broad-casting	no	backbone	PCTE database	no	no	views and update records

table 5.1. Multiple view support of different frameworks, programming environment generators, and software development environments.

From this table some important features required for the MViews model can be identified. These include:

- Program structure and semantic representations which have a comparable generality and abstractness to those of LOGGIE and Mjølner abstract syntax tree-based environments while retaining the flexibility of GARDEN and Unidraw objects.
- Multiple view representations that use the same structural (and possibly semantic) representation as programs, as supported by Unidraw and the ItemList.
- Program and view editing mechanisms appropriate to the kind of view rendering being used, as supported by FIELD and GARDEN.

- Consistency management for program and view updates supporting efficient semantic attribute recalculation, program-to-view and view-to-program update propagation, and incremental view updating, as supported by the ItemList and ODG.
- Change recording against program and view components to support documentation of program changes.
- A generic, extensible undo/redo mechanism, as supported by PECAN, Unidraw and the ItemList.
- A program and view persistency mechanism with a level of abstraction comparable to GARDEN and Mjølner environments.
- Tool integration and extensibility mechanisms comparable to that of FIELD environments.

5.3.   An Overview of MViews

We have developed MViews to satisfy most of the requirements of environments discussed in Section 5.1. In the development of MViews we have aimed for a homogeneous solution to providing these environment facilities which allows different requirements to be satisfied in a consistent, reusable manner based on a uniform conceptual model of programming environments.

5.3.1.   Program Graphs

Any program can be represented as a directed graph (Arefi et al 90). An incomplete program can be represented as a collection of disjoint directed graphs or a directed graph with "unexpanded nodes" (Arefi et al 90), similar to alternate choice or compound abstract syntax grammar nodes (Minör 91, Reps and Teitelbaum 87). MViews represents programs as a collection of (possibly disjoint) directed graphs, called *program graphs*. Program components are represented as *elements* (graph nodes) and are connected by *relationships* (labelled graph edges). Fig. 5.1. shows an example of an MViews program graph for part of the drawing program from Chapter 4.
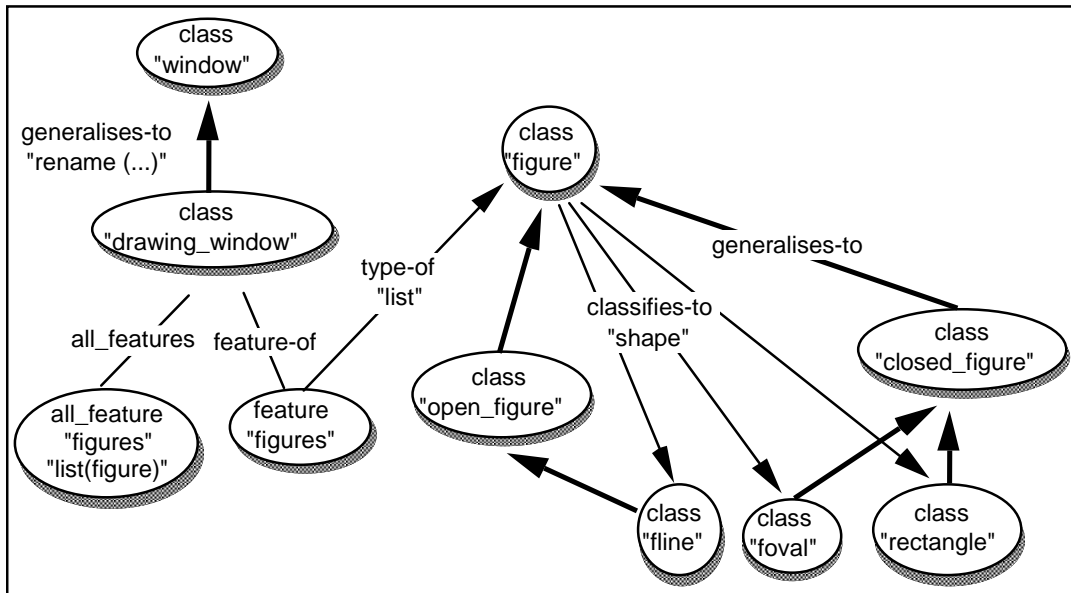
fig. 5.1.   Part of the program graph for the drawing program.

Each element and relationship represents a specific kind of *program component* and can have *attributes* (name/value pairs) associated with it (illustrated as unnamed, quoted text in fig. 5.1.). Some relationships are simple in that they just link related elements while others contain information about the relationship (for example, renamed features for generalisations and classifier name for classifications).

A program graph is a dependency graph. When a component is modified (added, updated or deleted) other related components (connected to the modified component via relationships) are notified of the exact change to the updated component. These may in turn be modified, depending on the language structure and semantics for the program under construction.

Components, attributes and relationships can be used to represent static language semantic values, in addition to structural, syntactic values. For example, the interface for a class (i.e. all its feature names and their types) might be represented by a group of `all_feature` elements, as shown in Fig. 5.1. When a feature of the class is updated, or features are added and deleted, the class will be notified of this change. The class can then respond to this feature change by updating the `all_features` semantic information appropriately (by recomputing it entirely or performing an incremental update). Language-specific semantic constraints can be achieved in a similar fashion by responding to update notification. For example, features of a class must have unique names and when a feature is renamed, the feature's class will be notified of this change. The class will respond to this feature change notification by checking that the new feature name is indeed unique (if not, the feature rename can be reversed and an error flagged).

As program structures are typically made up of nodes and labelled edges (Arefi et al 90) representing programs via graphs is very general. It is appropriate for most of the

program structures an environment should model including graph-based visual languages (Backlund et al 90). The advantages over plain abstract syntax structures is support for the modelling of graph-based languages and flexibility of construction (program graph components may be built up independently and then combined via appropriate relationships).

5.3.2. Views and View Components

All program graphs in MViews are grouped by *views*. MViews defines three types of view:

- the *base view* is a canonical representation of a complete program. There is one base view per program and all information about a program's structure and semantics held by an environment is stored as program graphs in this base view.
- *subset views* represent subsets of a base view and may overlap so the same information can be accessed and manipulated via different subset views. Users determine the composition of a subset view (i.e. the base components it views) and add, modify or remove subset components interactively. Examples of systems incorporating a similar notion to subset views include:
  - Ispel (Grundy et al 91), where multiple views describe overlapping subsets of a base view of an object-oriented program.
  - The dynamic and static views of MELD (Kaiser and Garlan 87) which partition programs into respectively overlapping and non-overlapping subsets.
  - Database views which filter out unwanted information. Database views are usually non-updateable, limiting the consistency management problems (although see (Horowitz and Teitelbaum 86, Langerak 90)).
- *display views* 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, including PICT (Glinert and Tanimoto 85), PECAN (Reiss 85), GARDEN (Reiss 87), and Ispel (Grundy et al 91). Users interact with display views to modify either graphical figures and connectors or textual characters which are translated into subset and base view modifications.

MViews programs (base views) are a collection of program graphs. Subset views of a base view may be constructed which are also program graphs. Subset view graphs represent sub-graphs of the base view graphs and a subset view may contain one or more disjoint graphs. A subset of the components and relationships in the base graph is represented in the subset view (i.e. base view elements and relationships have corresponding subset view

elements and relationships). The subset view's components (subset components) are subsets of the base view's components (base components) they represent[7].

Subset components are usually connected to base components via relationships. As a subset view is defined as being a partial view of the base view information, modifying a subset component is defined by MViews to be the same as modifying the base component it is linked to. Similarly, modifying a base component means all the subset components linked to this base component are modified in the same way[8]. For example, if a subset class is renamed, the base class the subset class is linked to is renamed. This results in all subset classes of the base class being renamed.

Relationships between base components and subset components (called subset/base relationships) allow changes to be propagated bi-directionally between a base component and its subset components. To maintain view consistency when it is updated a subset component translates updates on itself into appropriate updates on its base component. Similarly, when a base component is updated its subset components are notified of this change. These subset components interpret the change and modify their own state to be consistent with that of their base component.

Subset components need not always be connected (mapped) to a base component. This allows partial, but controlled, inconsistency at the view level. It also provides a mechanism for retaining subset view components when their base has been deleted so programmers can determine the change to make to a view (remap the subset component to another base component, delete the subset component and possibly related components, or otherwise change the composition of the subset view). View-specific information such as font details can also be represented in this way as unmapped subset components. Subset components typically model one base component although they may

---

[7]i.e. a subset view component may define a subset of its base view component attributes and relationships.

[8]This base->subset and subset->base translation may not always occur, as subset components can hold subset view-specific information that is not described in the shared base view. For example, if a subset component holds font information, changing this would not affect the subset's base component. Similarly, changing base component information that the subset component does not view (i.e. that the subset component is not interested in) will not require any modification to the subset component. For example, if a feature's type is changed but a subset component of the feature does not use this type value, the subset component need not be updated.

be mapped to more than one base component[9]. Base components typically have more than one subset component in one or more subset views.

Each subset view is rendered (displayed) either graphically or textually using an appropriate display view. A display view component (display component) renders a subset component in a textual or graphical form. Display components are re-rendered when their subset component changes and updates on a display component are translated into subset component updates by editing operations or dialogues.

Fig. 5.2. shows some typical base, subset and display view components and their relationships for an SPE-like environment.
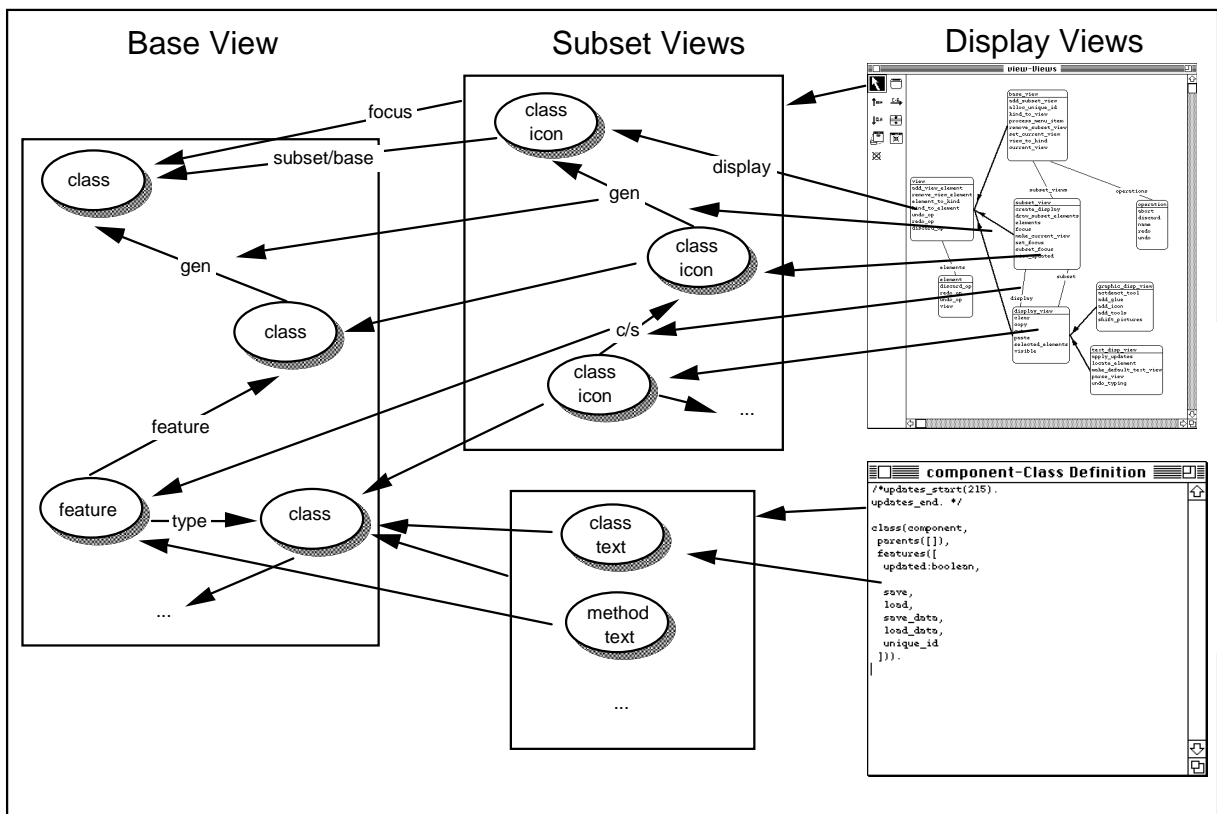
fig. 5.2.   Typical program and view storage in MViews.

In this example, the base view is composed of classes and their relationships. Two subset views are provided which represent sub-graphs of the base program graph. These subset views are displayed, one graphically, as class icons and generalisation and client-supplier glue, and one textually, as class and method text. Each subset component is linked to its

---

[9]This allows composite components to be represented in subset views. For example a "feature icon" for SPE might have a class and feature name and be mapped to the base class and base feature at the same time (so it can respond to changes in both base components).

corresponding base component via a subset/base relationship and each display to its subset. Base components can have more than one subset component but display components render only one subset component (though a subset component may represent more than one base component). Subset views conceptually focus on one base component and are thus "owned" by this base component. The owning base component for a subset view is designated the *focus* of the subset view.

Object dependency graphs are also used to represent subset views. This allows the same kinds of structures and the same kinds of manipulations to be performed on subset views as those used for base program data. It also allows the relationship between a base component and its subset(s) to be expressed in terms of a dependency relationship. An advantage of this is that changes to a base component can be sent to its subset components and changes to a subset component can be sent to its base components in the same manner (avoiding a deficiency of the MVC and Unidraw models).

5.3.3. Operations and Update Records

Graph *operations* are employed to modify program graphs. 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. Components can be added and deleted, attributes fetched and updated, relationships established and dissolved, and views created with display components added to or removed from them. Textual views can be typed and parsed to cause base changes while graphical figures can be dragged, edited interactively, selected and deselected, and so on.
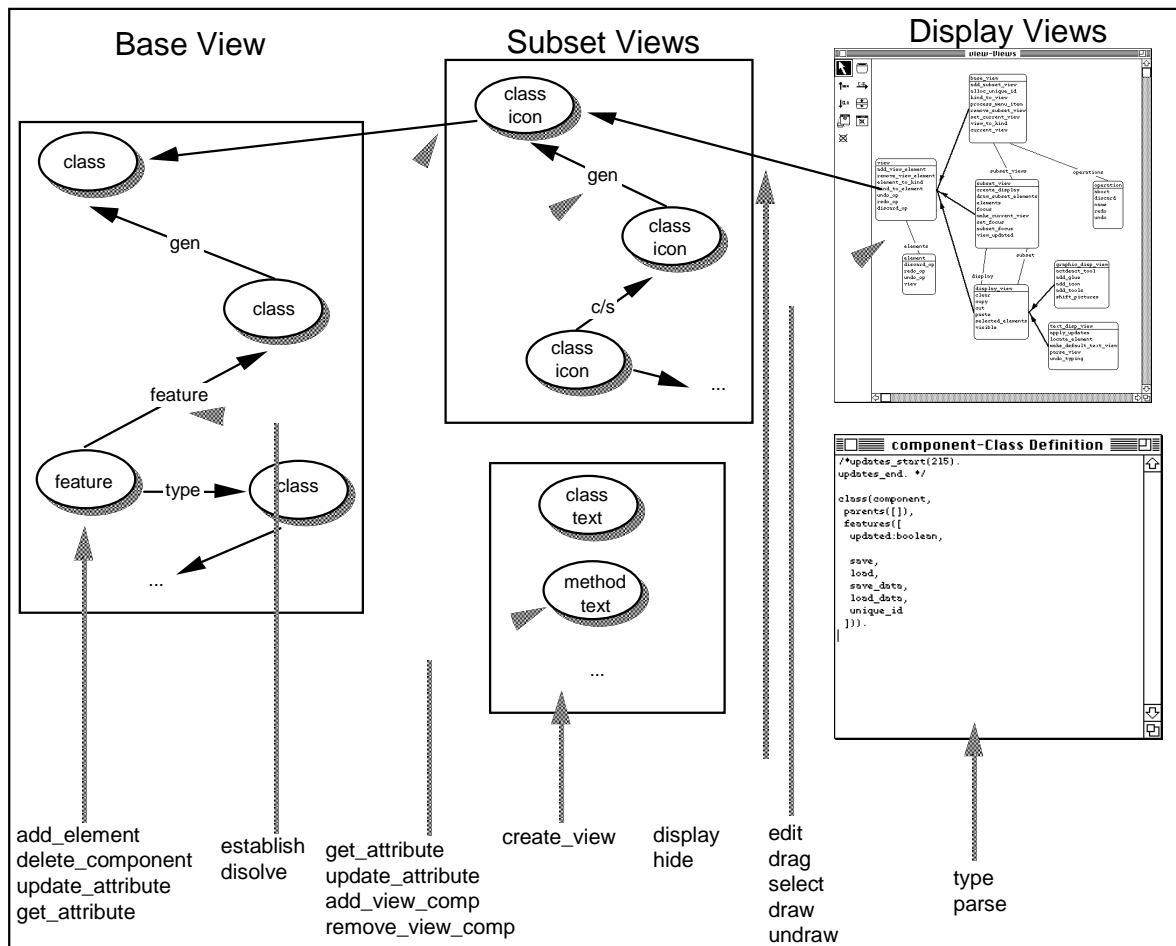
fig. 5.3.   Some typical operations affecting MViews program graphs and their views.

Sections 4.5. and 5.1. discussed the need to manage program changes in environments. MViews supports change recording and propagation using *update records* generated by applying graph operations. A component records each change made to itself by an operation as an update record, which is, conceptually, a sequence of values of the form:

        <Component,UpdateKind,Value$_1$,...,Value$_n$>

where:

- `Component` is the updated component
- `UpdateKind` describes the kind of update that has been carried out
- `Value`$_1$,...,`Value`$_n$ are additional `UpdateKind`-specific values describing the exact change that took place

Fig. 5.3. shows some of the updates that can be applied to components of an MViews system. Base and subset component operations generate update records, and Table 5.2. describes the fundamental program graph operations, the update records these operations generate and what the operation/update record means.

Operation	Update Record	Description

update_attribute(Comp,Attribute, New)	update_attribute(Comp,Attribute, Old,New)	Update attribute Attribute of component Comp and set its value to New (Old = old attribute value)
add_element(Kind,NewEl)	add_element(NewEl)	Add a new element of type Kind
establish(Kind,Parent,Child,NewRel)	establish(Kind,Parent,Child)	Establish a relationship of type Kind between components Parent and Child
dissolve(Kind,Parent,Child)	dissolve(Kind,Parent,Child)	Dissolve the relationship of type Kind between components Parent and Child
delete_component(Comp)	delete_component(Comp)	Delete component Comp
create_view(Kind,NewView)	create_view(NewView)	Create a new subset view
add_view_comp(View,Comp)	add_view_comp(View,Comp)	Add a component Comp to a view View
remove_view_comp(View,Comp)	remove_view_comp(View,Comp)	Remove a component Comp from a view View
record_update(Comp,UpdateRecord)	UpdateRecord	Record an update record UpdateRecord against a component Comp (i.e. propagate UpdateRecord to Comp's dependents)
store_update(Comp,UpdateRecord)		Store update record UpdateRecord against component Comp (doesn't generate any update record)

Table 5.2. A summary of the fundamental MViews operations and update records.

To document the changes it has undergone, a component may store update records against itself using a list attribute. Update records may be stored in an application-specific form for environment designer's convenience. For example, a base class in SPE may store the changes it has undergone since being created to document its modification history. An update record of the form `update_attribute(Feature, feature_name, OldName, NewName)` might be stored as `rename_feature(OldName, NewName)` against the base class.

Every component has zero or more related components that may be affected by a change to itself (called *dependent components*). For example, an SPE class may be dependent on its generalisation class (which it inherits features from) and the features it defines (as these determine if the class as a whole has been modified). In addition, a base component's dependents include its subset components and a subset component's dependents includes its base component(s). Components send any update records generated by updates to themselves to these dependent components.

Dependents interpret updates and modify themselves (if necessary), possibly generating further update records. Updates can be directly applied to display components to reflect changes to their base component. Alternately, update records may be expanded into a human readable form and rendered with a display component (for example, unparsing update records and displaying them with textual view components in SPE). The second approach is useful for subset views where it may not be possible to directly apply the update to the view's components (for example, after the addition or deletion of a client-supplier relationship in SPE). Update records may also be stored for use when undoing or redoing a change. To cause an undo of the last editing operation the update records associated with the operation (i.e. all the updates generated by it) can be sent back to the components that created the updates for reversal. Similarly, a reversed operation can be redone by sending the updates to their creators for reapplying.

MViews uses update records to record (document) program changes, propagate change to dependent components via relationships, maintain textual and graphical view consistency to the base, and define an undo/redo facility. They could also support data-driven and lazy semantics calculation (in a similar manner to that of (Wilk 91)) by transitive dependencies (one component dependent on another by way of a third component) and by temporarily storing updates until required for lazy application.

Update records provide several advantages over `Notify`/`Update` and state transition propagation. As an update record documents the exact change a component has undergone dependents of the component can update themselves in a very incremental manner. For example, a display component can determine the exact change to one of its base's attributes or a change in a related display component. This can be used to implement efficient incremental update and redrawing of the display from the base. For example, if a feature name is removed from a base attribute list a subset component viewing this base component's list does not need to reconcile its state to the base component completely, like Unidraw, and does not need to completely redisplay or update itself, which may be required with an MVC model. This mechanism can also provide constraints or incremental updates between related subset components. For example, if an icon is dragged its sub-icons can move their position or reconfigure themselves based on the change to their parent without the whole structure needing to be redisplayed, such as may be required with other approaches (Wilk 91). Update records can also be stored against components to document component change and provide lazy recalculation of attribute values.

Base and subset component updates can be propagated to each other using this mechanism without the need for intervention from a controller/editor component. This allows a more modular approach to the propagation of updates. Subset components are modified in the same way as base components and detect these updates (by being dependent on themselves) and propagate them to their base. As MViews uses relationships to determine dependency, update records do not need to store additions and deletions of dependents like Wilk's change reports (as these are recorded by `establish` and `dissolve` relationship operations as update records). Unlike Dannenburg's `ItemList` structure, MViews can represent a much richer set of structures using elements and relationships and as updates are recorded sequentially, redundant copies of `Item` (attribute) values are not stored, resulting in a simpler undo/redo mechanism.

5.3.4. Program Editing using Views as Tools

Software development environments provide tools to modify programs and to perform other management tasks (Meyers 91, Reiss 90b). MViews environments use tools associated with a display view to modify subset graphs (and thus, indirectly, base program graphs). Base information can also be accessed or modified directly by sending

operations to the base view or its components. For example, a subset component can access base component information not contained in its subset view or base component operations it doesn't supply.

Graphical view editors are structure-oriented, providing tools for manipulating specific aspects of a program, and utilise a direct manipulation interface to modify renderings of subset components (graphical display component structures). These modifications are translated into subset component operations by the editing tools supplied by MViews or by graphical display components in an application-specific manner. Subset components then update base components by interpreting these updates on themselves and applying operations to their base components.

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. Parsers generate a parse tree which is then given to each subset component in the view. A textual subset component compares its parse tree to the base program state and required changes to the base are computed and applied to reflect changes made to the textual view. Individual text elements can also be structure-edited using menu commands. Text and graphic editors can be tailor-made for an application or specialised from generic MViews tools.

Users interact with MViews systems either via textual and graphical display views or by using menus and dialogues. Dialogues can access subset and base information and update this information directly.

Tool extensibility and integration can be supported by MViews environments using display views and dialogues (which give a consistent user interface) and subset views (which can provide a tool-specific interface to a cannonical program structure stored in the base view). Subset views can provide a data mapping facility for exporting and importing external tool data using parsing and unparsing in a similar manner to textual display views and the tool mapping facilities of ICAtect (Amor et al 91). Display views and dialogues can be used to provide tool user interface integration in a similar manner to FIELD (Reiss 90b). Update records could also be used for tool communication as they equate to the events used in selective broadcasting (Reiss 90a).

5.3.5. Program Persistency

MViews components can be saved and reloaded from persistent storage by converting attribute and relationship values into a persistent form and vice-versa. This process is usually application-specific with MViews providing save and load operations that write and read persistent data respectively. Components need only be saved if they have been updated since their previous reloading and could be incrementally reloaded when

required (i.e. when accessed their persistent form found and converted into component form, then added back into the program graph).

5.3.6. Summary

MViews models programs as program graphs and groups these base program graphs using a base view. Subset views of a base view are also modelled as program graphs with each subset view component being a partial view of one or more base components. Program graphs are object dependency graphs with changes to a component being propagated to dependent components in the graph. This mechanism provides a uniform model for base/subset view modelling, propagating updates between components and base/subset components, and allows documentation and undo/redo facilities to be provided by storing generated updates. Display views and dialogues provide an interaction mechanism for users which allow subset view components to be rendered and updated (which in turn may update the base view). Textual view components are rendered as text and are parsed to produce base updates. Graphical views are composed of figures which are interactively edited with tools and dialogues.

5.4. MViews Specification Language

In this section a simple language is developed called MViews Specification Language (MVSL). This language captures the important abstractions proposed for the MViews model in Section 5.3. It also provides a preliminary analysis tool and documentation tool for describing the important aspects of MViews-based programming environments. MVSL is illustrated with examples from IspelM, a generalisation of SPE for constructing and visualising object-oriented programs using multiple textual and graphical views.

5.4.1. Rationale for MVSL and MVisual

Our initial work with Ispel (Grundy et al 91) indicated a need for two distinct specification techniques for programming environments that support multiple views: a description of the state of a program and the editing semantics for this program state; and a description of a user's perception of the environment in terms of interactions with views of the program (Grundy 91). These two specifications can then be used together to perform an analysis for new environments, document existing environments, and be used when extending environments (to ensure extensions are both consistent and well-defined). MVSL attempts to provide an abstract specification language to address the first issue based on the MViews model of environments.

MVSL

Given the model for MViews in Section 5.3. we require a specification language for describing such environments. Component kinds can be classified to a basic set of

component types (base views, base elements and relationships, and subset views, elements and relationships). Component attributes and relationships between components need to be referred to by name and given types. A predefined set of operations on program graphs may need to be augmented by sequences of operations for different applications. Components must respond to update records generated by other components they are dependent on by executing operations to change their state in response to these update records.

Initially we attempted to define MViews using an Object-Z (Duke et al 91) specification. MViews environments were specified by specialising a set of Object-Z classes and providing additional application-specific class attributes and operations. This approach, however, was based on the abstractions of an object-oriented architecture for MViews introduced in Chapter 6. The resulting specification still seemed too detailed for abstractly defining the basic state and semantics for an MViews environment. In particular, relationships and response to updates, two fundamental aspects of MViews, were much more obscured than in MVSL definitions. An Object-Z specification would, however, be suitable for formally specifying the object-oriented architecture of Chapter 6. In contrast, MVSL was developed to provide a specification based directly on the abstractions described in Section 5.3.

MVisual

In addition to specifying the state and modification semantics of this state with MVSL we require a mechanism for specifying the visual appearance and editing semantics of an environment. Display views and dialogues are used to interact with MViews environments and MVisual is used to specify the appearance and semantics of these visual entities. MVisual graphically illustrates the appearance of display views, display view components and dialogues using example-based programming and a form of visual programming. These techniques specify the effect of interactive manipulations on user interface entities and the changes these entities undergo in response to updates from MVSL subset views and components or other MVisual entities.

Update records generated by MVSL are assumed to be sent to MVisual which defines responses to susbet component changes graphically. Update records generated by MVisual are assumed to be sent to MVSL and are translated into update records sent to MVSL components. Thus update records are used to propagate changes between MVisual and MVSL as well as between MVSL components and MVisual components. MVisual is described in Section 5.6.

Formal Specification of MVSL

To show that the basic concepts of MViews are well-defined, an operational semantics specification of MVSL is given in Section 5.5. This uses a state based on the basic abstractions of MViews and specifies the effect on this state of applying the basic operations defined by MVSL. Application-specific operations and component responses to update records are composed of these basic operations and are hence well-defined in terms of their effect on the MViews program state. Communication with MVisual is assumed to be in a functional manner using streams of update records representing inputs (from MVisual) and outputs (from MVSL).

Implementing MViews Environments

Neither MVSL or MVisual specifications are sufficient for generating environments. Both languages lack sufficient power of expression for detailed descriptions of environments. MVSL does not provide sufficient programming language constructs and modularization for full environment specification and interfacing to existing tools. MVisual does not specify every special case response to user interaction and provides no detail about synchronising its interaction with MVSL. Both languages provide abstract specification techniques which can be used to analyse an environment in terms of the MViews model and document MViews environments. Chapter 6 proposes an object-oriented architecture for MViews environments which allows new environments to be constructed by specialising classes based on the fundamental abstractions of MViews. Chapter 7 uses this architecture as the basis for an implementation of MViews in Snart.

5.4.2. Overview of IspelM

IspelM defines an environment for constructing object-oriented software using multiple textual and graphical views with view consistency. Fig. 5.4. shows the basic abstractions used by IspelM. These abstractions include:

- Base elements: base clusters, which group related classes; base classes, which store information about classes for an object-oriented language; and base features, which store information about class attributes, methods, inherited features and deferred features. These base elements are connected by various relationships: generalisation, which specifies one class is generalised to another; client-supplier relationships, which indicate a class is used by another; and class and feature ownership, which indicates which cluster a class is owned by and which class a feature is owned by respectively.

- Class diagram subset views contain subset components of base components that are to be rendered graphically. Subset components are: class icons, which are partial views of base classes (class name and class feature names); generalisation glue, which is a view of base generalisation relationships; and client-supplier

glue, which views both base features (aggregate client-supplier relationships) and base client-supplier associations. Textual code views represent a textual rendering of base components. Subset components for textual code views are: class text, which represents a class's interface, part of a class interface or class documentation; and method text which represents a method interface and implementation or method documentation.

- A class diagram display view renders class diagram subset views and components in a graphical form. Users interact with this rendering using tools (direct manipulation) or dialogues (directly update subset view data). Textual code display views render textual code subset views as text and users edit this text interactively. Updated text is then parsed and the updates sent to textual subset components which update base components appropriately.
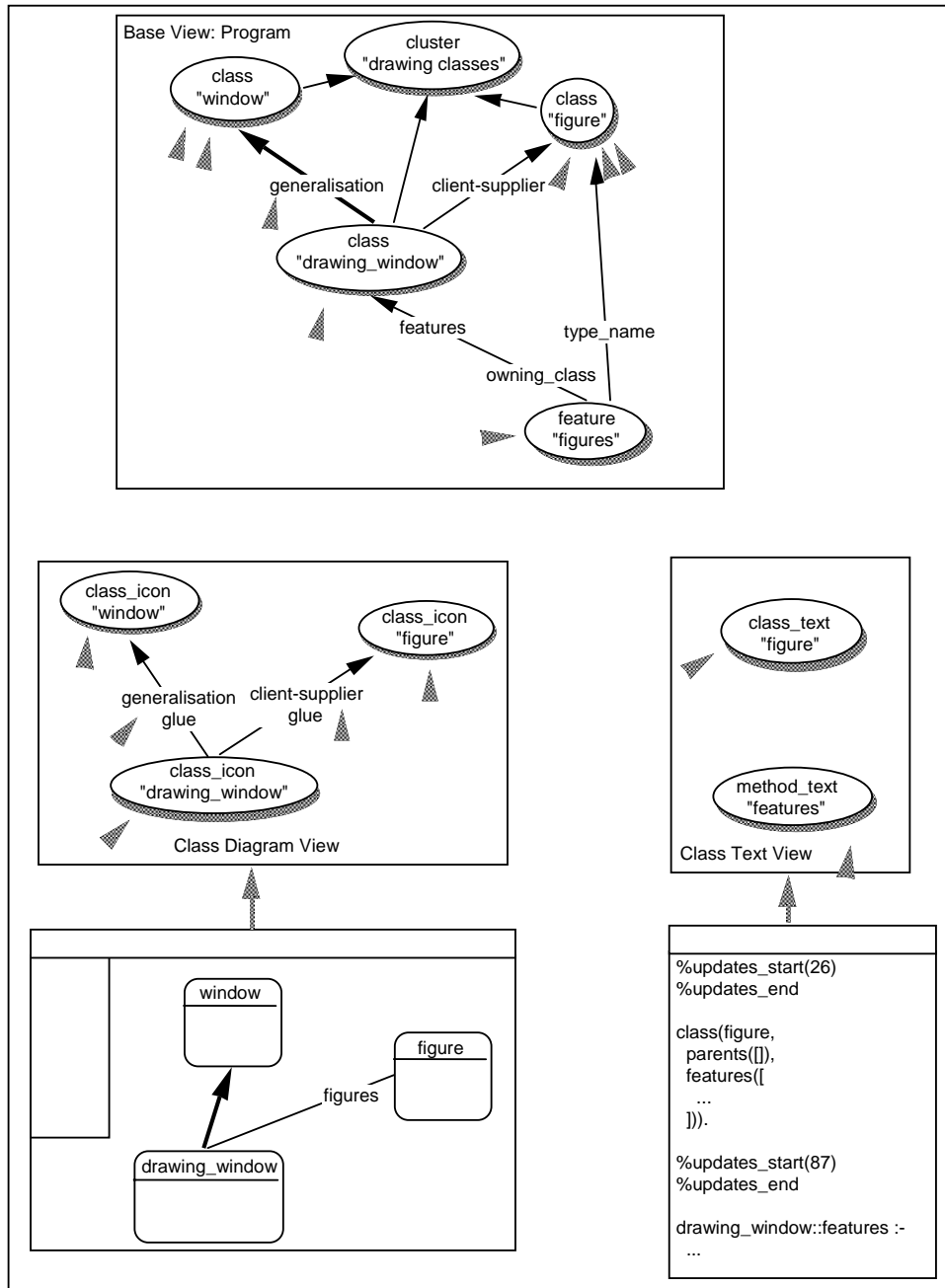
fig. 5.4.   Basic abstractions of IspelM.

SPE can be thought of as a specialisation of IspelM for programming Snart. Chapter 8 describes an architecture and implementation for IspelM and SPE based on the MViews architecture and implementation from Chapters 6 and 7.

5.4.3. Basic Abstractions of MViews

From the discussion of MViews in Section 5.3. the following basic abstractions can be made. An MViews program, *P*, is represented by the 4-tuple:

$$P \stackrel{\text{def}}{=} <C,A,R,V>$$

where:

- *C* is the set of components that comprise the program, made up of *<Component ,ComponentKind>* values, where *Component* is of type **Integer** (unique for *P)* and *ComponentKind* is of type **String**;

- *A* is the set of attribute triples of the form *<Component,AttributeName,Value>*, where *AttributeName* is of type **String** and *Value's* type is one of **Boolean**, **Integer**, **String**, **List**(Value), or *Enumerated*, where *Enumerated* is of the form *<Value$_1$,...,Value$_n$>* and *Value$_i$* is of type **String**;

- *R* is the set of relationships between components, represented as triples of the form *<Relationship,Parent,Child>*, where *Relationship*, *Parent* and *Child* types are all of type *Component*;

- *V* is the set of views, as triples of the form *<Component,Focus,Elements>*, where *Focus* (of type *Component*) is the owning component of the view and *Elements* is a sequence of (possibly disjoint) *Component* values represented in the view[10].

For some program, p, C(p) = all the *Component* values for p; R(p) are all the relationship component values for p; and V(p) all the view component values for p. For some relationship, r, Parent(r) = the parent *Component* for r and Child(r) = the child *Component* for r.

The dependents of a component are all those components related to it, i.e.:

$$Dependents(p,c) \stackrel{\text{def}}{=} \{r \mid r \square R(p) \square (Child(r)=c \Delta Parent(r)=c)\} \approx$$
$$\{d \mid \exists r: r \square R(p) \square d=Child(r) \square c=Parent(r)\} \approx$$
$$\{d \mid \exists r: r \square R(p) \square d=Parent(r) \square c=Child(r)\}$$

Update records are represented as value lists of the form *<Component,UpdateKind,V$_1$,...,V$_n$>* where UpdateKind : **String**. A program, *P$_1$*, is manipulated by successively applying a sequence of operations *<O$_1$,...,O$_n$>* to *P$_1$* to form *P$_{n+1}$*. Any operation, *O$_i$*, applied to a component, *C$_j$*, produces a new state, *P$_i$'*, and sends an update record, *U$_o$*, to the dependents of *C$_j$*. Further operations may be generated by dependent component interpretation of these update records to produce the final state,

---

[10]We could describe the *Elements* and *Focus* of a view using relationships but this view representation helps for purposes of abstraction and reasoning with MViews program graphs.

P_{i+1}. The visual representation of any views, *{V1,...,Vm}*, containing C_j (or one of it's dependents, if updated by U_0) will be re-rendered to reflect the changed program state.

5.4.4. Defining Component Kinds with MVSL

Section 5.4.3. defines a conceptual representation for an instance of a program stored in an MViews environment. Each component has a *ComponentKind* associated with it which describes the "type" of the component and component attribute values also have associated types. To specify the state of all programs a particular MViews environment can represent we can use this notion of component and attribute types to produce a component specification for the environment.

Fig. 5.5. shows an example component specification from IspelM defining two component kinds, class and generalisation. Base component types are differentiated into *base element* and *base relationship* which together comprise the components of a base program graph. Base element types define base program graph nodes while base relationship types define base program graph edges. *Attributes* for a component are defined by `attributes` and *relationships* the component needs to access by `relationships`. The significance of `operations` and `updates` is described in the following sections. Appendix F describes the complete concrete syntax for MVSL.

```
-- The base element class
--
base element class
   attributes
      class_name : string
      kind : [normal, abstract]
   relationships
      cluster : one-to-one cluster
      generalisations : generalisation.child
      client_suppliers : client_supplier.parent
      classifiers : classifier.parent
      features : one-to-many feature
      specialisations : one-to-many class
      all_features : one-to-many component all_feature
   operations
      ...
   updates
      ...
end class

-- Base generalisation relationship
--
base relationship generalisation
   parent class
   child class
   relationships
      renames : one-to-many rename
   operations
      ...
   updates
      ...
end generalisation
```

fig. 5.5.   An example of basic state definitions for MVSL.

Attributes are declared as AttributeName : AttributeType, with allowable types being integer, boolean, string, list(Type), and [enumerated_value$_1$,...,enumerated_value$_n$]. Attributes can also be declared as the same type as an attribute of some other component using AttributeName1 : like Component.AttributeName2. For an instance of a component kind, the value of one of the component instance's attributes must be of the same type as the attribute is declared as in MVSL. A relationship component declares, in addition to its attributes and relationships, distinguished parent and child attributes (as shown for generalisation in fig. 5.5.). These are declared with component types which determine the component kinds the relationship connects. For example, generalisation declares its parent and child to be classes in fig. 5.5.

All inter-component connections are described in terms of relationships. A component declares the relationship components it is interested in accessing as RelationshipName : RelComp, where RelComp is a relationship component name. The component declaring the relationship must be either the parent or child of the relationship component and this is

indicated by appending a `.parent` or `.child` qualifier to each relationship component name. For example, `classifiers` for a `class` in fig. 5.5. are declared as `classifiers : classifier.parent`, which indicates that a `class` instance is interested in all `classifier` relationships to itself where it is the parent (as classes are also the children for a group of classifier relationship instances).

Relationship names for a component refer to the relationship components connected to an instance of the defining component. For example, the value of `classifiers` for some instance of `class` from fig. 5.5. will be all classifiers connected to the class instance where the class instance is the parent of the classifier relationships.

Relationships may also be declared as simple link relationships, of the form `RelationshipName : one-to-one CompKind` or `RelationshipName : one-to-many CompKinds`, where `CompKind` is a component kind name. This is a short-hand declaration included so simple component linking relationships (which define no attributes or relationships of their own) needn't be defined as relationship components. The value of such a relationship name for a component instance is either the connected component instance (for one-to-one relationship links) or a list of component instances (for one-to-many links). For example, the value of `features` for an instance of `class` from fig. 5.5. is a list of `feature` instances connected to the class by any `class.features` link relationships where the class instance is the parent for the features relationship.

A *base view* component is used to group base elements and relationships. Only one base view may be defined per MVSL specification. Fig. 5.6. shows the definition for a base view for IspelM. An IspelM base view, as represented in fig. 5.4., maintains relationships to the clusters and classes that comprise an object-oriented program. These are defined by `clusters` and `classes` one-to-many relationships in fig. 5.6. Appendix D gives a complete MVSL specification for IspelM.

```
-- Program for IspelM
--
base view program
   attributes
      name : string

   relationships
      clusters : one-to-many cluster
      classes : one-to-many class

end program
```

fig. 5.6.   A base view defined by IspelM.

5.4.5. Subset Views and Components

Subset components represent subsets of the base view's components and are grouped by subset views. MVSL allows *subset elements*, *subset relationships* and *subset views* to be declared, as shown in fig. 5.7.

```
-- Class icons represent class name/kind and arbitrary
features (as their names)
-- owned by base class.
--
subset element class_icon
   attributes
      class_name : like class.class_name
      kind : like class.kind
      feature_names : list like feature.feature_name

   relationships
      view : one-to-one class_diagram_view
      base : one-to-one class

end class_icon

-- Generalisation Glue
--
subset relationship generalisation_glue
   parent class_icon
   child class_icon
   attributes
      name : string

   relationships
      view : one-to-one class_diagram_view
      base : one-to-one classifier

end generalisation_glue

-- Class text
--
subset element class_text
   attributes
      class_name : like base_class.name

   relationships
      view : one-to-one class_code_view
      base : one-to-one class

end class_text
```

```
-- Class diagram view
--
subset view class_diagram_view
   components
      class_icon, generalisation_glue, cs_or_feature,
classifier_glue

   attributes
      name : string

   relationships
      focus : one-to-one class

end class_diagram_view
```

fig. 5.7.   Subset views and subset view components from IspelM.

Subset elements and relationships can define the base components they view and subset view they are contained in using relationships. Subset views define the subset components they can contain by `components Elements`. In fig. 5.7., `class_icon` and `generalisation_glue` define their state to be a subset of their base component state. `class_icon` could also define relationships such as `generalisations : generalisation_glue` and `client_suppliers : client_supplier_glue` if these are to be accessed from `class_icon`. These relationships correspond to those defined in fig. 5.4. for IspelM subset views and components. For example, MVSL defines a `class_icon` to be linked to its base class by relationship `base` and to its view by relationship `view`, which corresponds to the relationships in fig. 5.4. for `class_icon`. Similarly `generalisation_glue` is linked to its base generalisation and subset view, and `class_diagram_view` is linked its focus base class.

Composite subset components can be defined by having a subset component dependent on two or more base components using multiple relationships. Client-supplier glue is defined in this way so it can view a base feature (if its an aggregate client-supplier relationship) or base client-supplier (if its a feature call or argument/local variable association). Subset components can also be related to one another to produce subset component dependencies. Subset components receive updates from base components and can transform these into operations on themselves. Subset components can also transform updates on themselves into operations on their base components.

5.4.6. Operations

Instances of programs defined using MVSL need to be manipulated, which corresponds to a program being constructed and changed. Components need to be added and deleted

and component attributes changed. Given the fundamental component kinds used by MVSL, a basic set of *operations* can be defined, as described in table 5.3.

Operation	In Arguments	Out Arguments	Description
add_element	CompKind	ComponentID	create a new element component
delete_component	ComponentID		deletes any component from the MViews program graphs
get_attribute	ComponentID, Attribute	Value	fetch component's attribute value
update_attribute	ComponentID, Attribute, NewValue		update component's attribute value
establish	CompKind, Parent, Child or CompKind.RelName, Parent, Child	ComponentID	establish a relationship between two components
dissolve	CompKind, Parent, Child or CompKind.RelName, Parent, Child		dissolve relationship between two components
create_view	CompKind	ComponentID	create new view component
add_view_component	CompKind, ComponentID		add a component to a view
remove_view_component	CompKind, ComponentID		remove a component from a view
store_update	ComponentID, List(Value)		store an update against a component
record_update	ComponentID, List(Value)		record an update against a component (propagate to dependents)

table 5.3. Basic operations for MVSL.

Elements are added using `add_element` and components deleted using `delete_component`. Attributes are fetched and updated with `get_attribute` and `update_attribute`. An alternative syntax for attribute fetch (`get_attribute`) is `Component.Attribute(Value)` and for attribute update (`update_attribute`) is `ComponentID.Attribute:=Value`. Relationships are established and dissolved with `establish` and `dissolve` and views created with `create_view`. View components are added and removed using `add_view_component` and `remove_view_component`. Update records can be propagated to dependents using `record_update` and stored against components using `store_update`[11].

While the set of operations from table 5.3. is sufficient for modifying program graphs, additional operations can be defined which use these basic operations. These *component-specific operations* allow "macro-operations" to be defined which provide application-specific operations useful for allowing reuse of common sequences of program graph changes. Procedural-style *commands* are also defined which support conditional execution of operations and looping. These are shown in table 5.4.

---

[11]Update records are stored as lists of values and the distinguished component attribute `updates : list(list(Value))` is used to hold these stored update records for each component.

Command Syntax	Description

if <boolean expression> then
    <operations if true>
else
    <operations if false>
end if
	if-then-else statement. The else-part is optional.

while <boolean expression true> do
    <operations>
end while
	while statement. Loops through operations while the expression remains true.

forall <variable> on <list> do <operations> end forall	forall statement. Interates through all values in list and executes operations with variable set to each list value in turn.

table 5.4. MVSL procedural commands.

Variables are used to hold non-component related values. Arguments and local variables used in component-specific operations have types similar to component attributes. Arguments are defined to pass values to an operation (designated by the prefix `in`) and/or return values produced by an operation (designated by `out`). Global values can also be defined and an initialisation operation is provided to define the initial state for an MViews environment (as shown in fig. 5.8). The scoping of MVSL operations is similar to methods in most object-oriented programming languages with the attributes, relationships and operations defined by a component referred to by name only inside the defining component's operations. Examples of component-specific operations from IspelM are shown in fig. 5.8. To apply these operations to a component the syntax `Component.Operation(Argument_1,...,Argument_n)` is used.

```
-- The base element class
--
base element class
   ...
   operations
      -- add feature
      --
      add_feature(in name : like feature.feature_name,
         in kind : like feature.kind,
         in type : like feature.type_name,
         out new_feature : feature) is
         add_element(feature,new_feature)
         new_feature.init(kind,type)
         establish(class.features,self,new_feature)
      end add_feature
   ...
      -- Map this class icon to a base class
      --
      map(in do_map : boolean) local
         base_class : class
      is
         base_class := program.find_class(class_name)
         if base_class \== nil then
            if do_map then

   establish(subset_relationship,base_class,self)
         end if
end class
```

```
-- Class icon.
--
subset element class_icon
   ...
   operations
      -- reselect new class
      --
      reselect_class(in name : like class.class_name)
      local
         other_class : class
      is
         other_class := program.find_class(name)
         if other_class \== self then
            dissolve(subset_relationship,base,self)
            class_name := name
            remap
         end if
      end reselect_class
   ...
end class_icon

-- Global values
--
program : program
   -- base view reference

-- Initial computation
--
initialise
   add_element(program,program)
   program.record_update(init)
end initial
```

fig. 5.8.   Some component-specific operations from IspelM.

The value of an expression is defined by MVSL in a similar manner to most programming languages. Operators include addition and subtraction for integers and boolean algebra. In addition to arguments and local variables, component-specific operations define a distinguished local variable `self` which allows an operation to determine the component it is being applied to. *Functional operations* can be defined for a component which are component-specific operations that return a value. A functional operation is declared of the form `OpName(...Arguments...) : Type is ... end OpName`. Functional operations refer to their result using a distinguished variable `result` (in a similar manner to Eiffel functions).

Relationships can be established between a component and a "nil component" value, indicated by the value `nil`, which allows the actual relationship to be created but later

reestablished to an actual component[12]. A `nil` value can be returned by a function and is the default value for all the attributes of a component when it is first created.

MVSL currently permits a procedural environment specification where variables can be assigned a value (in addition to component attributes being assigned a value) and operations and commands are assumed to be applied in sequence to a program state. A functional specification could be used where variable names define values that can not be assigned to. The procedural `forall` and `while` commands could be replaced with mapping functions and recursive functions respectively. Operations must be sequential as each successive operation produces a new program state from the program state produced by its prior operation. This could be described by functional composition where, for a program state p and two operations f and g, the final program state is g(f(p)). A concrete representation of this might be f ; g where f and g are operations.

5.4.7. Update Operations

When a component's state is changed by an operation (i.e. its attributes or relationships modified or it is deleted) it must broadcast this change to any components dependent on its state with an appropriate update record. Dependent components then interpret this update record and apply further operations to reconcile their state to that of the changed component (to maintain a consistent program state). Components define *update operations* to process update records sent to them by other components, as shown in fig. 5.9.

---

[12]This is useful when only partial information for determining the components to relate is supplied the relationship or the components to relate depend on attribute values or other relationship values for the relationship.

```
base relationship generalisation
   ...
   updates
      -- When establish/dissolve generalisations,
      -- maintain specialisations list attribute
      -- Store updates against owning_class, not
generalisation.
      --
      establish(rel:relationship, kind : string,
         parent : class, child : class) where
         rel = self and kind = "generalisation"
      is
         owning_class.store_update([add_gen,child,parent])
         parent.specialisations := parent.specialisations
++ {self}
      end establish

      dissolve(rel:relationship, kind:string, parent :
class, child : class) where
         rel = self and kind = "generalisation"
      is

   owning_class.store_update([remove_gen,child,parent])
         parent.specialisations := parent.specialisations -
- {self}
      end establish
   ...
end generalisation
```

```
subset element class_icon
   ...
   updates
      remap_feature(name, new_name:like
feature.feature_name,
         new_type:like feature.type_name,
         new_kind:like feature.kind,
         show:boolean) where true local
         feature : feature
      is
         feature := base.find_feature(new_name)
         if feature = nil then

   base.add_feature(new_name,new_type,new_kind,feature)
         end if
         remove_feature_name(name)
         if show then
            add_feature_name(new_name)
         end if
      end remap_feature

      -- Translate base attribute updates into subset
changes
      --
      update_attribute(class : class, name : string,
         old : string, new : string) where
         class = base and name = "class_name"
      is
         class_name := new
      end update_attribute
...
end class_icon
```

fig. 5.9.   Update operations from IspelM.

Updates are "guarded", input-only operations. They are only executed if their component receives an update record with the same name as the update operation, the same number and type of arguments, and if the expression guard for the update operation evaluates to true. This provides a simple pattern match algorithm for determining which update to apply. Update operations have input-only arguments (outputs do not make sense) and hence the `in` keyword is discarded for them.

Subset and base component updates are propagated to each other by subset components defining update operations to detect updates to their base component and to themselves. Subset components detect base component updates and transform them into appropriate operations on themselves (if necessary). A subset component also detects updates on itself and transforms these into base component updates (if appropriate).

We currently assume the implementation language for MViews provides its own persistency model (such as an object store or database). Chapters 7 and 10 discuss the issue of program persistency in further detail.

5.5. A Formal Specification of MVSL

5.5.1. Operational Semantics

Operational semantics provide a mechanism for specifying programming language static and dynamic semantics using validity and meaning functions. Given a program construct, validity functions provide a boolean result indicating whether the construct is valid or not. Given a program execution "state" and a program construct, meaning functions return a new state which is defined to be the effect of "executing" the construct in the old state. Reviews of operational and denotational semantics can be found in (Tennent 76, Gordon 79). MVSL was specified using operational semantics and then, to verify the correctness of this specification, we implemented the operational specification using Gofer (Haskell) (Jones 92). (Finlay and Allison 93) provides an example of the usefulness in verifying a formal specification via an implementation using a functional language.

An abstract syntax for MVSL is defined which allows us to represent the structure of MVSL programs without the additional syntactic sugar used in the MVSL concrete syntax. Identifiers defined by MVSL are associated with type values which describe the type of attributes, relationships and operations. The state of an MViews program is described by a tuple which represents the views and program graphs stored by an environment. The basic operations and commands for MVSL are defined as functions which map an initial state and operation to a new state, hence defining the effect of executing the operation. Two forms of update records are used in this formal specification. One form is used to broadcast changes between program components and these components interpret these update records with update operations. The other form is used to communicate with MVisual and is part of the program state. Whenever an update record is generated by a program component a corresponding "output" update record is recorded in the program state to inform MVisual of the component change.

5.5.2. Concrete vs. Abstract Syntax

MVSL programs, as described in Section 5.4., contain much syntactic sugar that hides the actual structure of an MVSL specification. It is convenient to be able to avoid such semantically irrelevant details by using an abstract form of syntax that specifies the structure of programs and not how they are represented as strings of symbols (Tennent 76). Fig. 5.10. shows a concrete syntax for an MVSL program and its equivalent abstract syntax (using Gofer notation).

| base element class
    attributes
       class_name : string
       kind : [normal,abstract]
    relationships
       features : one-to-many feature
       generalisations : generalisation.child
    operations
       add_feature(in name : string, in type : string,
          out new_feature : feature)
       is
          add_element(feature,new_feature)
          new_feature.init(name,type)
          establish(class.features,self,new_feature)
          establish(feature.owning_class,
             new_feature,self)
       end add_feature
end class | BaseElement "class"
    [Attribute "class_name" StringType,
     Attribute "kind" Enum ["normal","abstract"]]
    [Relationship "features" OneToMany "feature",
     Relationship "generalisations"
       (CompAttr "generalisation" "child")]
    [Operation "add_feature"
      [InArg "name" StringType,
      InArg "type" StringType,
      OutArg "new_feature" CompType "feature"] Void
      [AddElement "feature" (Ident "new_feature") :&
      ApplyOp (FuncOp "new_feature" "init" [])
        [Ident "name",Ident "type"] :&
      EstablishLink (CompAttr "class" "features")
       (Ident "self") (Ident "new_feature") :&
      EstablishLink (CompAttr "feature" "owning_class")
       (Ident "new_feature") (Ident "self")]] |
| Concrete Syntax | Abstract Syntax |

fig. 5.10. A concrete MVSL program example and its equivalent abstract syntax form.

Fig. 5.11. gives an example of an abstract syntax definition for MVSL using Gofer notation (abstract syntax productions are defined as user defined data types). Appendix C gives a full Gofer implementation for this operational semantics specification for MVSL. This abstract syntax definition for MVSL mirrors the basic abstractions made by MVSL for defining declarations (base views, elements, etc.), commands (operations and procedural control structures), expressions and types.

```
data Program = Pro [Decl] Command

data Decl = BaseView Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
    BaseElement Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
    BaseRelationship Ide ParentDecl ChildDecl [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
    GraphicView Ide [Ide] [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
    GraphicIcon Ide [Ide] [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
    ...

data AttributeDecl = Attribute Ide Type
data RelationshipDecl = Relationship Ide Type
data OperationDecl = Operation Ide [OpArgumentDecl] [LocalDecl] Command |
    Function Ide [OpArgumentDecl] Type [LocalDecl] Command
...

data Command = Exp := Exp |
    Eifthen Exp Command Command |
    Ewhile Exp Command |
    AddElement Ide Exp |
    DeleteComponent Exp |
    Establish Type Exp Exp Exp |
    Dissolve Type Exp Exp |
    Store Exp UpdateValue |
    ...

data Exp = IntLit Int |
    StringLit String |
    True_ | False_ |
    Ident String |
    Op Opr Exp Exp |
    FuncOp Exp Ide [Exp]

data Type = BoolType |
    StringType |
    IntType |
    ListType Type |
```

```
ComponentType Ide |
CompAttrType Ide Ide |
...
```

fig. 5.11. Some abstract syntax definitions for MVSL.

5.5.3. Declaration Types

Abstract syntax declarations define identifiers that refer to components and variables. These identifiers are used to determine the type of variables and structure of components when despatching component-specific operations, selecting an update operation to apply, and when matching update operation arguments and types. Thus a *type map* from identifiers to their type structure is required. This type map could also be used to determine the static validity of declarations, commands and expressions (for example, that the same component kind isn't defined twice, a component-specific operation actually exists for a component, and that elements of an expression have compatible types). Fig. 5.12. illustrates some of the type map definitions for the operational specification of MVSL.

```
-- Declaration value maps type identifier to type value
--
type DeclValue = Ide -> TypeValue

-- TypeValue
--
data BasicKind = KBaseView | KBaseEl | KBaseRel |
   KGraphicView | KGraphicIcon | KGraphicGlue |
   KTextView | KTextElement |
   KComp | KLinkRel

data TypeValue = TCompData BasicKind [Ide] CompTypes |
   TString |
   TInteger |
   TOneToMany TypeValue |
   TComp Ide |
   TCompAttr Ide Ide |
   ...

-- Component types map component attribute etc. names to CompType values
--
type CompTypes = Ide -> CompType

data CompType = CAttribute TypeValue |
   CRelationship TypeValue |
   COperation OpArgs TypeValue OpLocals CommandMeaning |
   CUpdates [CUpdate] |
   CNotDefined

-- Command/exp meaning: given some function which maps Commands/Exp to new States/Values,
-- return a State/Value given a State
--
type CommandMeaning = (State -> Command -> State) -> State -> State
type ExpMeaning = (State -> Exp -> Value) -> State -> Value

-- Arguments and local variables for operations
--
type OpArgs = [(Ide,InOrOut,TypeValue)]
data InOrOut = In | Out

type OpLocals = [(Ide,TypeValue)]
```

Fig. 5.12. Type and component declaration values for MVSL.

`DeclValue` is defined to be a type whose value is a function mapping identifiers to `TypeValues`. `TypeValue` is a user defined data type which specifies an identifier's type is a string, integer, one-to-one relationship, component identifier, component attribute, component definition, and various other types (as defined in Appendix C).

A component definition (`TCompData`) has a basic kind (one of base view, base element, subset relationship, etc.) which is defined by `BasicKind`. Components define a set of identifiers whose types are component values (attributes, relationships, operations and update operations). The type `CompTypes` is used to map component definition identifiers to component value types. The value of `CompTypes` is a function which maps component identifiers to their component value types (defined by `CompType`). A component definition also includes a list of all the identifiers defined for the component (i.e. the domain of `CompTypes` for the component).

Operations and update operations define arguments, a type (for functional operations), local variables and a command. An operation command does not have a type as such, but rather a meaning when the operation is executed. This is defined as the meaning of the command given a particular program state (defined as `CommandMeaning`), i.e. the effect of executing the operation command (the type of `State` is given in Section 5.5.5.). Similarly, an update operation guard expression has a meaning which is its value for a particular program state (the type of `Value` is defined in Section 5.5.5). Component-specific operations and update operations are despatched for components in a similar manner to methods for object-oriented languages. Thus the operational semantics specification for MVSL requires these command and expression meanings (and, in fact, the whole DeclValue for a program), in addition to a program `State`, to specify the dynamic semantics for an MVSL program.

5.5.4. Building a Type Map for MVSL Program Declarations

Given a list of abstract syntax component declarations for a program, a type map must be constructed for these declarations. Fig. 5.13. shows part of the definition for `program_decls`, a function which returns a type map (`DeclValue`) for a list of abstract syntax declarations (`[Decl]`) .

```
-- Compute the declarations value for a list of program declarations
--
-- Returns DeclValue for program and a list of identifiers  to create locations for (i.e. globals)
--
-- rel_comps computes any link relationships defined by the component
-- and adds a DeclValue for their name (name = CompKind.RelName)
--
program_decls :: [Decl] -> DeclValue -> [Ide] -> (DeclValue,[Ide])
program_decls [] dv gs = (dv,gs)
program_decls (d:ds) dv gs = (new_dv,new_gs) where
    (n,tv,rs,globals) = decl_value d
```

```
      (new_dv,new_gs) = program_decls ds
          (updateDeclValue (rel_comps dv rs n)  n tv) (gs++globals)

-- Function over-riding for DeclValue
--
updateDeclValue :: DeclValue -> Ide -> TypeValue -> DeclValue
updateDeclValue dv n tv i = if i==n then tv else dv i

-- Compute the Ide/TypeValue/Link Relationships/Globals for a component/global declaration
--
rel_comps :: DeclValue -> [RelationshipDecl] -> Ide -> DeclValue
rel_comps dv [] comp_name = dv
rel_comps dv ((Relationship name (OneToOne _)):rs) comp_name = new_dv where
  new_dv = rel_comps (updateDeclValue dv (comp_name++"."++name) link_rel) rs comp_name
rel_comps dv ((Relationship name (OneToMany _)):rs) comp_name = new_dv where
  new_dv = rel_comps (updateDeclValue dv (comp_name++"."++name) link_rel) rs comp_name

-- Compute the Ide/TypeValue pair for a declaration and any relationships it defines
--
decl_value :: Decl -> (Ide,TypeValue,[RelationshipDecl],[Ide])
decl_value (BaseView name as rs os us) = (name,tv,rs,[]) where
  (names,comp_ts) = default_types KBaseView (update_types (op_types (rel_types (attribute_types
      ([],emptyCompTypes) as) rs) os) us)
  tv = (TCompData KBaseView names comp_ts)
decl_value (BaseElement name as rs os us) = (name,tv,rs,[]) where
  (names,comp_ts) = default_types KBaseEl (update_types (op_types (rel_types (attribute_types
      ([],emptyCompTypes) as) rs) os) us)
  tv = (TCompData KBaseEl names comp_ts)
decl_value (BaseRelationship name pd cd as rs os us) = (name,tv,rs,[]) where
  (names,comp_ts) = default_types KBaseRel (update_types (op_types (rel_types (attribute_types
      (parent_types (child_types ([],emptyCompTypes) cd) pd) as) rs) os) us)
  tv = (TCompData KBaseRel names comp_ts)
...

-- Compute attribute types for list of attribute declarations
--
attribute_types :: ([Ide],CompTypes) -> [AttributeDecl] -> ([Ide],CompTypes)
attribute_types (names,ct) [] = (names,ct)
attribute_types (names,ct) ((Attribute n t):as) =
  attribute_types ([n]++names,(updateCompTypes ct n (CAttribute (type_value t)))) as

-- Compute relationship types for list of relationship declarations
--
rel_types :: ([Ide],CompTypes) -> [RelationshipDecl] -> ([Ide],CompTypes)
...

-- Compute operation types for list of operation declarations
--
op_types :: ([Ide],CompTypes) -> [OperationDecl] -> ([Ide],CompTypes)
op_types (names,ct) [] = (names,ct)
op_types (names,ct) ((Operation n arg_decls loc_decls command):os) =
  op_types ([n]++names,(updateCompTypes ct n (op_value arg_decls loc_decls (op_meaning command)))) os
op_types (names,ct) ((Function n arg_decls t loc_decls command):os) =
  op_types ([n]++names,(updateCompTypes ct n (fn_value arg_decls t loc_decls (op_meaning command)))) os

-- Compute update types for list of update declarations
--
-- This produces a list of guarded, input-only operations which are
-- event-driven by updates on a component.
--
update_types :: ([Ide],CompTypes) -> [UpdateDecl] -> ([Ide],CompTypes)
    update_types ([n]++names,(updateCompTypes ct n (CUpdates [upd_op]))) us
...

-- Value of an operation declaration
op_value :: [OpArgumentDecl] -> [LocalDecl] -> CommandMeaning -> CompType
op_value as ls command = (COperation (op_arg_types as) TVoid (local_types ls) command)

-- Value of a "functional operation" is same as for operation but with a type
fn_value :: [OpArgumentDecl] -> Type -> [LocalDecl] -> CommandMeaning -> CompType
...
```

```
-- Value of an "update operation" is same for operation but arguments are input-only
update_value :: [LocalDecl] -> [LocalDecl] -> ExpMeaning -> CommandMeaning -> CUpdate
...

-- Bind operation arguments to in/out status and type
--
op_arg_types :: [OpArgumentDecl] -> OpArgs
...

-- Bind local variables to type
--
local_types :: [LocalDecl] -> OpLocals
...

-- Bind update arguments to type
--
update_arg_types :: [LocalDecl] -> OpArgs
...

-- Value of an abstract syntax type
--
type_value :: Type -> TypeValue
type_value (StringType) = (TString)
type_value (IntType) = (TInteger)
type_value (OneToMany c) =(TOneToMany c)
type_value (ComponentType n) = (TComp n)
...

-- Default attributes and component types for a component given its "BasicKind"
--
data DefaultType = Default Ide CompType

default_types :: BasicKind -> ([Ide],CompTypes) -> ([Ide],CompTypes)
...
```

fig. 5.13. Constructing the DeclValue and CompTypes functions.

For each component or global abstract syntax declaration (Decl), the function `decl_value` returns the identifier used to refer to the component or variable, its TypeValue, any link relationship declarations for the component, and any global variables it defines. All link relationships (named as "`CompName.RelName`") have simple component definitions and the function `rel_comps` extends `DeclValue` with these definitions so link relationships can be treated in the same way as more complex component relationships.

Each attribute, relationship, etc. declaration for a component needs an identifier to `CompType` mapping (defined as `CompTypes` for a component definition `TypeValue`). The functions `parent_types`, `child_types`, `attribute_types`, `relationship_types`, `operation_types` and `update_types` return an identifier/`CompTypes` pair given a list of component value declarations. The type of an operation or update operation includes identifier to `TypeValue` mappings for the arguments and local variables defined by the operation, and a meaning for the operation command and guard expression.

5.5.5. State

MViews represents programs as object dependency graphs with subset views also being object dependency graphs. The state of an MViews environment will hence be similar to that described in Section 5.4.2. with components, attribute values, relationships and views.

Relationships and views can be modelled as components with attribute values corresponding to their `parent`, `child` and view `components` values.

In addition to this program and view state, an MVSL program state needs to map identifiers to corresponding values. An *environment* (Tennent 76) is used to map identifiers to denotable values (literals, pointers, and component values) while a *store* (Tennent 76) is used to map locations to expressible values (literal values: integers, strings, and component identifiers). This allows operation arguments and local variables to be defined as extending an environment (when executing the operation command) and the environment to be restored to its former state (after execution of an operation command).

Fig. 5.14. illustrates the state definitions for the operational specification of MVSL.

```
-- CompStore
type CompID = Int
type CompStore = CompID -> Ide -> CompValue
data CompValue = NoCValue | CValue Dv

emptyCompStore :: CompStore
emptyCompStore _ _ = NoCValue

new_comp :: CompStore -> Ide -> (CompStore,CompID)
new_comp s k = new_comp' 1 where
   new_comp' i = case s i "class" of
       NoCValue -> (updateCompStore s i "class" (Rv (Vstring k)),i) ; _ -> new_comp' (i+1)

updateCompStore :: CompStore -> CompID -> Ide -> Dv -> CompStore
updateCompStore s c a v i j = if i==c && j==a then (CValue v) else s i j

remove_comp :: CompStore -> CompID -> CompStore
remove_comp s c i j = if i==c then NoCValue else s i j

-- All relationships for a component are stored by "relationships"
comp_rels :: State -> CompID -> [CompID]
comp_rels s c = rels where
   rels = case comps s c "relationships" of
       (CValue (Rv (Vlist rel_values))) -> values_to_comps rel_values
        _ -> []

-- View components for a view are stored in "components"
view_comps :: State -> CompID -> [CompID]
view_comps s c = vcomps where
   (CValue (Rv (Vlist comp_values))) = comps s c "components"
   vcomps = values_to_comps comp_values

-- Denotable values
data Dv = Loc Location | Rv Value | CompValue CompID Ide

-- Expressable values
data Value = Vnum Int | Vbool Bool | Vstring String | Vcomp CompID | Vlist [Value] | Nil

-- Store/Location for state variables
type Location = Int
data ValueOrUnused = Used Value | Unused
type Store = Location->ValueOrUnused

new :: Store -> Location
updateStore :: Store -> Location -> Value -> Store

-- Environment for state variables
data ValueOrUnbound = Bound Dv | Unbound
type Env = Ide -> ValueOrUnbound
```

```
updateEnv :: Env -> Ide -> Dv -> Env

-- Update records = "term" of form Kind(Value1,Value2,...)
data UpdateRecord = UpdateRec Ide [Value]
data Output = OV CompID UpdateRecord

-- The MViews program state is a tuple with component and location stores, an environment
-- and output list.
-- State also stores the DeclValue for a program as operations and updates must be
-- despatched on a per-component basis (could pass this value to all functions using
-- State, but its easiest to put it here).
--
type State = (CompStore,Env,Store,[Output],DeclValue)

-- Update/query state elements
update_comps :: State -> CompStore -> State
update_comps (_,e,s,o,dv) c = (c,e,s,o,dv)
...
comps :: State -> CompStore
comps (c,_,_,_,_) = c
...
```

fig. 5.14. A program state for MVSL.

5.5.6. Commands, Operations and Update Operations

Expressions

Expressions are evaluated with respect to a given program state. An expression can be evaluated to a denotable value (for assignment and variable parameter arguments) or an expressible value (for use in computation). Fig. 5.15. illustrates the meaning functions for expressions.

```
-- Get the value (Value) of an expression (i.e. an rvalue)
--
rval :: State -> Dv -> Value
rval s (Loc l) = r where (Used r) = (store s) l
rval s (Rv v) = v
rval s (CompValue c a) = cv where
   (CValue (Rv (Vstring ct))) = (comps s c "class")
   (TCompData bk vs comp_types) = (declarations s) ct
   cv = case (comp_types a) of
       (CAttribute t) -> av where
           (CValue (Rv av)) = (comps s) c a
       (CRelationship t) -> (rel_value c s a t)
       (COperation [] t [] command) -> fn_result where
           ...

-- Get the denotable value for an expression (i.e. an lvalue)
--
exp_val :: State -> Exp -> Dv
exp_val _ (IntLit i) = Rv (Vnum i)
exp_val _ (StringLit s) = Rv (Vstring s)
exp_val s (Ident i) = ev where
   (Bound ev) = (env s) i
exp_val s (CompVal e a) = (CompValue c a) where
   (Vcomp c) = exp_rval s e
exp_val s (FuncOp c_exp arg_exps) = ev where
   (CompValue c a) = exp_val s c_exp
   (CValue (Rv (Vstring ct))) = (comps s c "class")
   (TCompData bk vs comp_types) = (declarations s) ct
   ev = case (comp_types a) of
       (COperation args t locs command) -> fn_result where
           ...
       _ -> (CompValue c a)
exp_val s (Op op lexpr rexpr) = opval op lv rv where
   lv = rval s (exp_val s lexpr)
   rv = rval s (exp_val s rexpr)
   opval Plus    (Vnum a) (Vnum b) = (Rv (Vnum (a+b)))
   opval Minus   (Vnum a) (Vnum b) = (Rv (Vnum (a-b)))
   ...
```

fig. 5.15. Expression meaning (including functional operations) for MVSL.

rval returns the expressible value given a denotable value. exp_val returns a denotable value given an expression. Literals are returned as a denotable form of their expressible value, identifiers return their denotable value in the current environment, operators are evaluated, and a functional operator returns a value (if an operation) or component value (if attribute or relationship). The value of a functional operation is defined in a similar manner to component-specific operations but also returns a value. Functional operations are currently defined to not alter the state of an MVSL program (as rval and exp_val do not return a new State). This restriction could be removed by returning a new State as well as a value for expressions (Tennent 76).

Expression and other meaning functions assume they are not given an invalid expression, command or operation for a given program DeclValue and State. Our current Gofer implementation returns a function exception if invalid abstract syntax values are given to meaning functions. This could be eliminated by performing static type checking for a given program (possibly when computing DeclValue) and not asking for the meaning of an invalid program. Another approach might use a continuation-style meaning function

specification (Tennent 76) which returns an "answer" (including error messages) for the whole program an expression or command is executed in.

Commands

To specify the dynamic semantics of an MVSL program, meaning functions for commands, operations and update operations are defined. The meaning of a command is the `State` returned after executing a command in a given program state. Fig. 5.16. shows example meaning functions for assignment and if-statements defined by the operational specification for MVSL.

```
-- Meaning of all commands
--
command_meaning :: State -> Command -> State
command_meaning s c@(l := r) = assign s c
command_meaning s c@(Eifthen e c1 c2) = if_then s c
command_meaning s c@(Ewhile e com) = while s c
...
command_meaning s c = operation_command s c

-- lv := rv
--
assign :: State -> Command -> State
assign s (lexp := rexp) = assign_result s lv rv where
  lv = exp_val s lexp
  rv = rval s (exp_val s rexp)

-- assign_result
--
-- if state variable => update store
-- if component attribute => update component store
--
assign_result :: State -> Dv -> Value -> State
assign_result s (Loc l) rvalue = update_store s (updateStore (store s) l rvalue)
assign_result s (CompAttr c a) rvalue = new_s where
  (CValue (Rv old_v)) = (comps s) c a
  updated_s = update_dependents s c (UpdateRec "update_attribute" [Vcomp c,Vstring a,old_v,rvalue])
  new_s = update_comps updated_s (updateCompStore (comps updated_s) c a (Rv rvalue))

-- if e then c1 else c2
--
if_then :: State -> Command -> State
if_then s (Eifthen expr if_command else_command) = new_s where
  (Vbool ev) = rval s (exp_val s expr)
  new_s = if ev then command_meaning s if_command
      else command_meaning s else_command
```

fig. 5.16. Command, assignment and if-statement meanings for MVSL.

The effect of assignment is to update the `Store` (if a variable) or `CompStore` (if a component attribute). Component attribute assignment equates to an update_attribute operation, which generates and propagates an update record using `update_dependents` (see below). A conditional statement evaluates its boolean expression and, if this expression evaluates to true, returns the `State` produced by executing its first command, or if false, the `State` produced by executing its second command.

Basic Operations

The meaning functions for operations return a `State` which is the effect of applying the operation to a component given an initial program state. Fig. 5.17. illustrates the meaning functions for the `add_element` and `establish` basic operations.

```
- Meaning of basic operation "commands"
--
operation_command :: State -> Command -> State
operation_command s c@(AddElement k e) = add_element s c
operation_command s c@(DeleteElement e) = delete_element s c
operation_command s c@(Establish kind p ch v) = establish_rel s c
...

-- add_element(in Kind,out CompID)
--
add_element :: State -> Command -> State
add_element s (AddElement kind new_var) = new_s where
   (comp_s,new_c) = add_component s kind
   (Loc new_loc) = exp_val comp_s new_var
   new_s = update_store comp_s (updateStore (store comp_s) new_loc (Vcomp new_c))

-- Add a new component and set given variable to the new component ID
--
add_component :: State -> Ide -> (State,CompID)
add_component s kind = (new_s,new_c) where
   (new_comps,new_c) = new_comp (comps s) kind
   (TCompData bk vs ct) = declarations s kind
   alloc_attributes [] cs c ct = cs
   alloc_attributes (n:ns) cs c ct =
        case (ct n) of
           (CAttribute t) ->
                if n == "class" then alloc_attributes ns cs c ct
                      else updateCompStore (alloc_attributes ns cs c ct) c n (Rv Nil)
            _ -> alloc_attributes ns cs c ct
   new_s = update_comps s (alloc_attributes vs new_comps new_c ct)

-- establish_rel(in Kind,in Parent,in Child,out NewRel)
--
establish_rel :: State -> Command -> State
establish_rel s (Establish kind parent child new_rel) = new_s where
   rk = rel_kind_type kind
   (Vcomp p) = rval s (exp_val s parent)
   (Vcomp c) = rval s (exp_val s child)
   (comp_s,new_r) = do_establish_rel s rk p c
   (Loc new_loc) = exp_val comp_s new_rel
   new_s = update_store comp_s
        (updateStore (store comp_s) new_loc (Vcomp new_r))

do_establish_rel :: State -> Ide -> CompID -> CompID -> (State,CompID)
do_establish_rel s rk p c = (new_s,new_r) where
   (r_s,new_r) = add_component s rk
   new_rs = updateCompStore (updateCompStore (comps r_s) new_r "parent" (Rv (Vcomp p)))
        new_r "child" (Rv (Vcomp c))
   new_pcr = updateCompStore new_rs p "relationships"
        (Rv (Vlist (comps_to_values([new_r]++comp_rels r_s p))))
   new_pcc = updateCompStore new_pcr c "relationships"
        (Rv (Vlist (comps_to_values ([new_r]++comp_rels r_s c))))
   updated_s = update_comps r_s new_pcc
   new_s = update_dependents updated_s c
        (UpdateRec "establish_rel" [Vstring rk,Vcomp p,Vcomp c])
```

fig. 5.17. The `add_element` and `establish` operation meaning functions.

Component-Specific Operations

Component-specific operations are applied to a component by allocating component, argument and local variables, executing the operation's command to produce a new `State`, and then deallocating the variables (by returning the initial `State` environment).

This final `State` is defined to be the meaning of the component-specific operation. Component-specific operations are defined to have a scope like object-oriented language methods and can access values defined by the component they are applied to. Fig. 5.18. illustrates part of the meaning function for component-specific operation application.

```
-- CompExp.OpName([ArgExp])
--
apply_operation :: State -> Command -> State
apply_operation s (ApplyOp exp arg_exps) = new_s where
   (CompValue c op) = exp_val s exp
   (CValue (Rv (Vstring ct))) = (comps s c "class")
   (TCompData bk vs cts) = (declarations s) ct
   arg_vals :: [Exp] -> State -> [Value]
   arg_vals [] s = []
   arg_vals (e:es) s = (arg_vals es s)++[(exp_rval s e)]
   new_s = case (cts op) of
       (COperation args t locs command) -> op_result where
           arg_vals = eval_args arg_exps s
           old_env = env s
           pre_op_s = alloc_self (alloc_locals (alloc_and_bind_args
                (alloc_comp_values s vs c) args arg_vals) locs) c
           post_op_s = dealloc_comp_values (dealloc_args (dealloc_locals
                (dealloc_self (command command_meaning pre_op_s)) locs) args) vs c
           op_result = update_env post_op_s old_env
       (CUpdates updates) -> apply_updates updates s c (arg_vals arg_exps s) vs
           -- call update operation as an operation

-- Evaluate lvalues for arguments
eval_args :: [Exp] -> State -> [Dv]
...

-- Allocate component values
alloc_comp_values :: State -> [Ide] -> CompID -> State
alloc_comp_values s [] c = s
alloc_comp_values s (n:ns) c = alloc_comp_values new_s ns c where
   new_env = updateEnv (env s) n (CompValue c n)
   new_s = update_env s new_env

-- Allocate & bind arguments for operation
-- In arguments have new location which is the Value of actual argument
--   (i.e. value parameters)
-- Out arguments have same Dv as actual argument
--   (i.e. variable parameters)
--
alloc_and_bind_args :: State -> OpArgs -> [Dv] -> State
alloc_and_bind_args s [] [] = s
alloc_and_bind_args s ((n,In,_):as) (v:vs) = new_s where
   rv = rval s v
   l = new (store s)
   new_store = updateStore (store s) l rv
   new_env = updateEnv (env s) n (Loc l)
   new_s = alloc_and_bind_args (update_store (update_env s new_env) new_store) as vs
alloc_and_bind_args s ((n,Out,_):as) (v:vs) = new_s where
   new_env = updateEnv (env s) n v
   new_s = alloc_and_bind_args (update_env s new_env) as vs

-- Allocate locals for operation
alloc_locals :: State -> OpLocals -> State
...

-- Allocate "self" variable for operation
alloc_self :: State -> CompID -> State
...

-- Deallocate a list of identifiers from Store
dealloc :: State -> [Ide] -> State
...
```

```
-- Deallocate "self" variable for operation
--
dealloc_self :: State -> State
...

-- Deallocate arguments for operation
--
dealloc_args :: State -> OpArgs -> State
...

-- Deallocate locals for operation
--
dealloc_locals :: State -> OpLocals -> State
...

-- Deallocate component values
--
dealloc_comp_values :: State -> [Ide] -> CompID -> State
...
```

fig. 5.18. The component-specific operation meaning function.

Update Operations

When an operation is applied to a component it generates an update record which is propagated to the component's dependents. These dependents test the update record against their update operations and execute any update operations which match the update record. A match is valid if the update operation has the same kind, number, and type of arguments, and its guard expression evaluates to true. Fig. 5.19. illustrates how this update propagation process is carried out by providing a meaning function `update_dependents`.

```
-- Dependents for a component are:
--  1) itself
--  2) all relationships it participates in
--  3) all other components its connected to via its relationships
--
dependents :: State -> CompID -> [CompID]
dependents s c = deps where
   rs = comp_rels s c
   deps = [c]++rs++collect_deps rs s c
   collect_deps [] s c = []
   collect_deps (x:xs) s c = cd where
       (CValue (Rv (Vcomp parent))) = comps s x "parent"
       (CValue (Rv (Vcomp child))) = comps s x "child"
       cd = if parent == c then [child]++collect_deps xs s c
            else [parent]++collect_deps xs s c

-- Send update record to dependents for a component
--
update_dependents :: State -> CompID -> UpdateRecord -> State
update_dependents s c u = new_s where
   update_dependents1 [] s _ = s
   update_dependents1 (d:ds) s u =
       update_dependents1 ds (update_from s d u) u
   output_s = update_output s ((output s)++[(OV c u)])
   new_s = update_dependents1 (dependents s c) output_s  u

-- Process update from another component
--
update_from :: State -> CompID -> UpdateRecord -> State
update_from s d (UpdateRec kind arg_vals) = new_s where
   (CValue (Rv (Vstring k))) = comps s d "class"
   (TCompData bk vs ct) = (declarations s) k
```

```
          new_s = case (ct kind) of
              (CUpdates updates) -> apply_updates updates s d arg_vals vs
              _ -> s

-- Apply an update to a component (if it supports the update)
--
-- Update operations are performed by finding a match (correct kind,
-- number and type of args and guard that evaluates to true) and
-- applying the operation as for component-specific operations
--
apply_updates :: [CUpdate] -> State -> CompID -> [Value] -> [Ide] -> State
apply_updates [] s d arg_vals vs = s
apply_updates ((UpdateOp args locs g command):us) s d arg_vals vs =
   if same_length_and_type (reverse args) arg_vals s
       then upd_s else apply_updates us s d arg_vals vs where
           vals :: [Value] -> [Dv]
           vals [] = []
           vals (v:vs) = (vals vs)++[Rv v]
           old_env = env s
           pre_op_s = alloc_self (alloc_locals (alloc_and_bind_args
               (alloc_comp_values s vs d) args (vals arg_vals)) locs) d
           upd_s = case (g exp_rval pre_op_s) of
               (Vbool True) -> op_result where
                   post_op_s = dealloc_comp_values (dealloc_self (dealloc_locals (
                   dealloc_args (command command_meaning pre_op_s) args) locs)) vs d
                   op_result = update_env post_op_s old_env
               _ -> apply_updates us s d arg_vals vs

same_length_and_type :: OpArgs -> [Value] -> State -> Bool
...
```

fig. 5.19. Update propagation for MVSL components.

5.5.7. Program Meaning

The meaning of an MVSL program is defined to be a sequence of output update records
the program generates given a set of input updates (assumed to be from MVSL). Fig. 5.20.
illustrates this meaning function for a program.

```
-- MVSL program meaning
--
data Input = IV Ide [Value]

-- Meaning of a Program is defined by its outputs given a set of inputs and definition
--
program :: Program -> [Input] -> [Output]
program (Pro decls command) i = out where
   (dv,gs) = program_decls decls emptyDeclValue []
   init_s = alloc_globals (emptyState dv) gs
   com_s = command_meaning init_s command
   out = output (run_program i com_s)

-- Need globals for program definition
--
alloc_globals :: State -> [Ide] -> State
...

-- Program is "run" by interpreting a sequence of "inputs" from MVisual
--
run_program :: [Input] -> State -> State
run_program [] s = s
run_program (i:is) s = new_s where
   new_s = run_program is (apply_input_update i s)

-- Translate input "update" record into operation on a component
--
```

```
-- Conceptually, MVisual generates these updates in respose to user interaction
-- MVSL's outputs are interpreted by MVisual which then updates view renderings
-- to indicate program change
--
apply_input_update :: Input -> State -> State
apply_input_update (IV "update_attribute" [Vcomp c,Vstring name,new]) s =
    assign_result s (CompValue c name) new
apply_input_update (IV "add_element" [Vstring kind]) s = new_s where
    (new_s,_) = add_component s kind
apply_input_update (IV "delete_component" [Vcomp c]) s =
    do_delete_component s c
...
```

fig. 5.20. The meaning of an MVSL program.

5.5.8. MVSL Programs

Using this Gofer implementation of the operational specification for MVSL, programs can be "executed" to produce outputs. Fig. 5.21. shows the concrete syntax for a simple MVSL program and Fig. 5.22. a corresponding abstract syntax for the program and the output update records produced when this program is "executed". The write operation, which generates an output update record, is introduced to illustrate the order of update operation application.

```
p : program
new_class : class
new_icon : new_icon

base view program
    attribute name : string
    relationship classes : one-to-many class
end program

base element class
    attributes name : string
    operations
        print_name is write self.name end print_name
        add (in lv : integer, rv : integer) : integer local
            temp : integer is
            temp := lv + rv
            result := temp
        end add
    updates
        update_attribute(comp : class, name : string, old : string, new : string) where
            name = "name" is
            write new
        end update_attribute
end class

subset element class_icon
    attributes
        name : string
        x : integer
    relationships
        view : one-to-one class_diagram
        base :one-to-one class
    updates
        update_attribute(class : class, cname : string, old : string, new : string) where
            cname = "name" is
            if name \== new then
                name := new
            end if
        end update attribute
end class icon
```

```
subset view class_diagram
   components class_icon
end class_diagram

initialise is
   create_view(program,p)
   add_element(class,new_class)
   add_element(class_icon,new_icon)
   establish(class_icon.base,new_class,new_icon)
end initialise
```

fig. 5.21. Concrete syntax for a simple MVSL program.

This program defines a base view `program`, a base element `class`, a subset element `class_icon`, and a subset view `class_diagram`. The initial state for the environment is one which contains a `program`, `class` and `class_icon` (with the `class_icon` a "subset" of the `class`). When the `class.name` attribute is updated, `class_icon` is informed of the change and updates its own `name` attribute to reflect the change to its base class (using an update operation for `update_attribute`). In the example in fig. 22., `class.name` is updated by an input update record which causes `class_icon` to update its own `name` attribute value. `class_icon.x` is also updated by an input update record. The operations applied to the MVSL program state produce corresponding output update records.

```
MVSL Program:
--
-- MVSL test program
--
test1 :: [Output]
test1 = output where
 output = program (Pro
   [(Global "p" (ComponentType "program")),
    (Global "new_class" (ComponentType "class")),
    (Global "new_icon" (ComponentType "graphic_icon")),
    (BaseView "program" [Attribute "name" StringType]
       [Relationship "classes" (OneToMany "class")] [] []),
    (BaseElement "class" [Attribute "name" StringType] []
       [(Operation "print_name" [] [] (EWrite (CompVal (Ident "self") "name"))),
        (Function "add"
           [InArg "lv" IntType,InArg "rv" IntType] IntType
           [Arg "temp" IntType]
           ((((Ident "temp") := (Op Plus (Ident "lv") (Ident "rv")) :&
           ((Ident "result") := (Ident "temp")))))]
       [(Update "update_attribute"
           [Arg "comp" (ComponentType "class"),
            Arg "name" StringType,
            Arg "old" StringType,
            Arg "new" StringType]
           (Op Eq (Ident "name") (StringLit "name"))
           []
           (EWrite (Ident "new")))]),
     (SubsetElement "class_icon"
       [Attribute "name" StringType,
        Attribute "X" IntType]
       [Relationship "view" (OneToOne "class_diagram"),
        Relationship "base" (OneToOne "class")] []
       [(Update "update_attribute"
           [Arg "class" (ComponentType "class"),
            Arg "cname" StringType,
            Arg "old" StringType,
            Arg "new" StringType]
           (Op Eq (Ident "cname") (StringLit "name"))
           []
           ((Eifthen (Op Neq (Ident "name") (Ident "new"))
```

```
            ((Ident "name") := (Ident "new"))
            (Eskip))))]),
    (SubsetView "class_diagram" (Components ["class_icon"]) [] [] [] [])]
  ((CreateView "program" (Ident "p")) :&
  (AddElement "class" (Ident "new_class")) :&
  (AddElement "class_icon" (Ident "new_icon")) :&
  (EstablishLink (CompAttrType "class_icon" "base") (Ident "new_class") (Ident "new_icon"))))

Inputs to program:
   [(IV "update_attribute" [Vcomp 2,Vstring "name",Vstring "NewName"]),
    (IV "update_attribute" [Vcomp 3,Vstring "X",Vnum 10])]

Output of function query "test1":
  [OV 3 (UpdateRec "establish_rel" [Vstring "class_icon.base", Vcomp 2, Vcomp 3]),
   OV 2 (UpdateRec "update_attribute" [Vcomp 2, Vstring "name", Vstring "OldName", Vstring "NewName"]),
   OV 0 (UpdateRec "write" [Vstring "NewName"]),
   OV 3 (UpdateRec "update_attribute" [Vcomp 3, Vstring "name", Nil, Vstring "NewName"]),
   OV 3 (UpdateRec "update_attribute" [Vcomp 3, Vstring "X", Nil, Vnum 10])]
```

fig. 5.22. Abstract syntax for MVSL program and its output update records.

5.6. Specification of Visual Appearance and Semantics

5.6.1. Rationale for MVisual

MVSL describes the base and subset level of an MViews system. To describe the display and user interaction aspects a graphical specification called MVisual is used. This separation of descriptions allows programmers to define an MViews environment in two steps: the first describes the state of an MViews environment using MVSL; the second describes the user interaction and display views using MVisual. A graphical specification for user interaction was chosen as it provides a more natural and expressive representation although it uses a somewhat less rigorous notation (including example-based definitions), similar to the PARTS instance-based programming system (LaLonde and Pugh 93). The two formalisms conceptually interact by passing update records. Updates generated by MVisual are translated into operations or update operations on MVSL components and MVSL updates are sent to MVisual for interpretation.

5.6.2. MVisual Fundamentals

Fig 5.23. illustrates the fundamental specification components of MVisual. Each MVisual definition is contained in a "View" which has a name and zero or more arguments. MVSL components are referred to by name (possibly an argument name for their enclosing view). The appearance of icons, glue, views and dialogues (referred to as *visual entities*) is defined graphically. For example, `Component Name` from fig. 5.23. is an icon made up of various graphical figures.

MVSL components are referred to by their name (possibly an argument name) in an oval, other MVisual views are referred to by name in a rectangle, and visual entities by a picture relating to their appearance (possibly containing a name for clarity). Additional information can be specified by pointer indicators (for example, click-area names, the

MVSL component value(s) a visual component value represents, and so on). Click-areas on icons are highlighted by a grey rectangle which can also be used to highlight aspects of a specification for clarity, naming or preciseness. Alternative visual forms can be specified by two or more pointers from the same place.
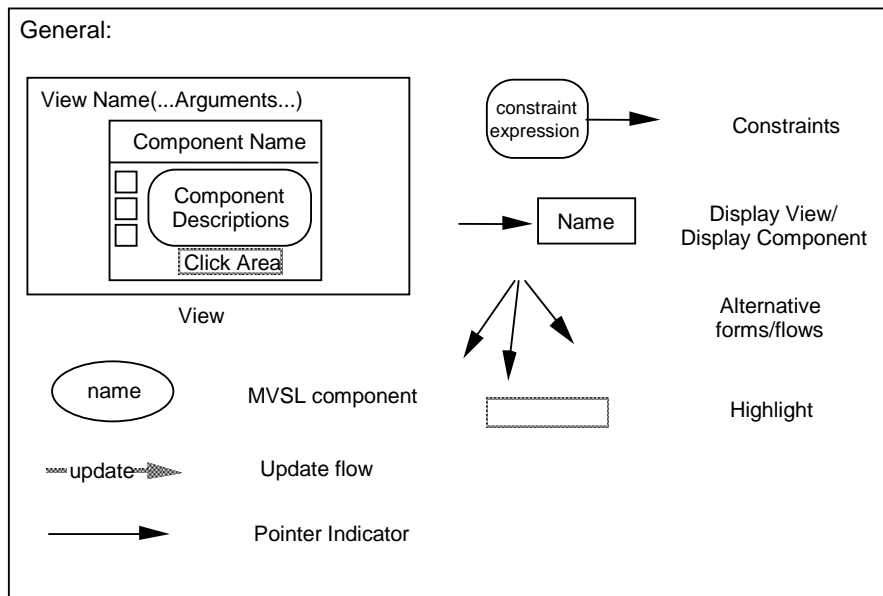
fig 5.23.   Fundamental visual components of MVisual.

Updates correspond to events which are either user interactions, MVSL update records or MVisual updates. Update flow between visual entities, MVSL components and MVisual views is specified by grey arrows annotated with an update name and (possibly) argument values. An update may flow to an MVSL component which defines a user interaction to be the application of an update operation to the MVSL component. Alternately, the update can flow to a view name (which indicates the view is displayed), or to another update flow (indicating the user interaction update generates a new update which in turn is sent to further components). Constraints can be added against appearance components or update flows to indicate conditions that must be satisfied for updates to be sent.

5.6.3. Icons and Glue

Appearance

Fig. 5.24. shows an example view defining the visual appearance for MVSL class icons. Appendix E contains a full specification of IspelM using MVisual.
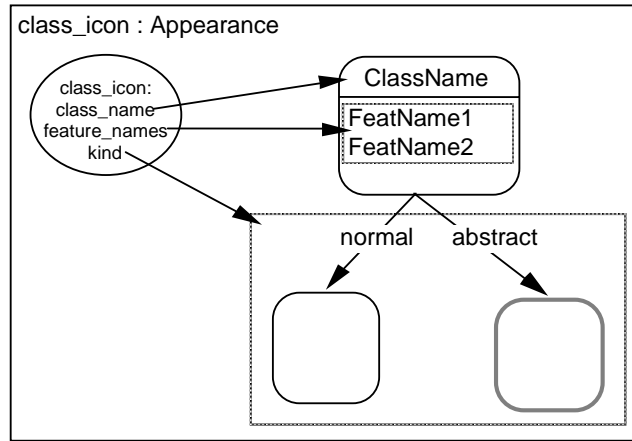
fig. 5.24. A visual appearance for MVSL class icons.

The `class_icon : Appearance` view defines the visual appearance for an MVSL `class_icon` (as defined in Section 5.4.) and a generic class_icon is represented in the oval. Some attribute values of class icon (`class_name`, `feature_names` and `kind)` correspond to values in the visual specification for a class icon, shown as it appears in SPE (see Chapter 4). The `ClassName` text value for the visual appearance is derived from `class_icon.class_name` (indicated by a pointer from the MVSL `class_icon` attribute value). `FeatName1` and `FeatName2` represent an example `feature_names` list from `class_icon` and are grouped to indicate they are derived from the same MVSL attribute. The `kind` of a `class_icon` determines the border for the visual appearance of a class icon. This is specified by an alternative pointer indicating which border appearances are used for different values of `class_icon.kind`.

Fig. 5.25. shows the appearance of client-supplier glue for MVSL `cs_or_feature` subset relationships. Multiple views are used to avoid clutter and constraints (inside oval-cornered rectangles) indicate values of MVSL attributes used to determine appearance. Class icons are indicated by a short-hand appearance with `Client` and `Supplier` names (to illustrate the relationship the client-supplier glue represents).
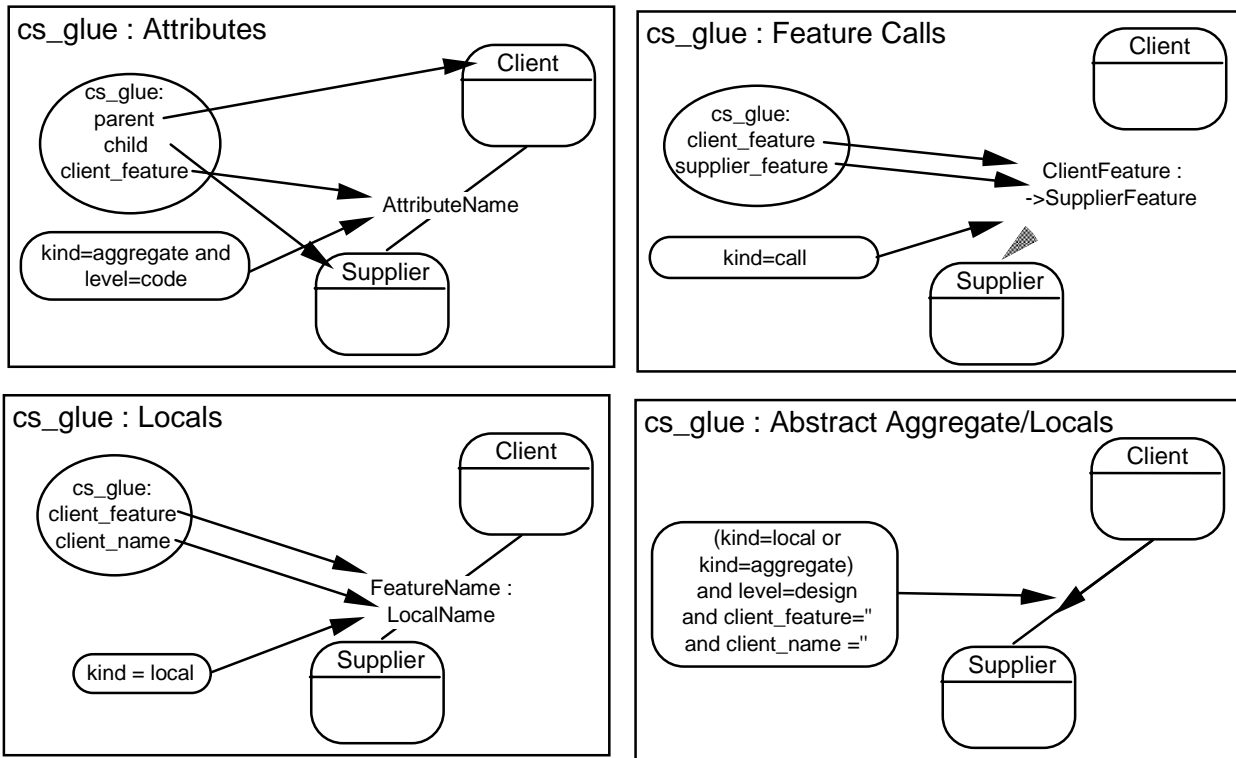
fig. 5.25. Visual appearance of MVSL `cs_or_feature` subsets.

Interaction

MVisual continues the MVSL theme of update record propagation between components and models user interaction by update propagation between visual entities and MVSL components. When a user interaction (described as an update record) is applied to some MVisual entity further update records may be generated which define the effect of the user interaction. These additional updates are then sent to MVisual entities and/or MVSL components (possibly different from the one the original update was applied to). Fig. 5.26. shows the effect of double-click updates on class icons.

A class icon defines "click areas" which determine the action taken if a double-click occurs within the click area (shown as shaded boxes inside the class icon in fig. 5.26.). Double-clicking on the left side of a class icon will open a dialogue for view or feature selection (indicated inside a square "view" box with the dialogue name and arguments). Double-clicking on the right side of a class icon class or feature name will display the `default_text_view` for the class or feature clicked on. MVisual assumes suitable values for `default_text_view` and the `display` update operation are defined for components and textual views (as they are common operations), but for preciseness these could be defined in MVSL and MVisual.
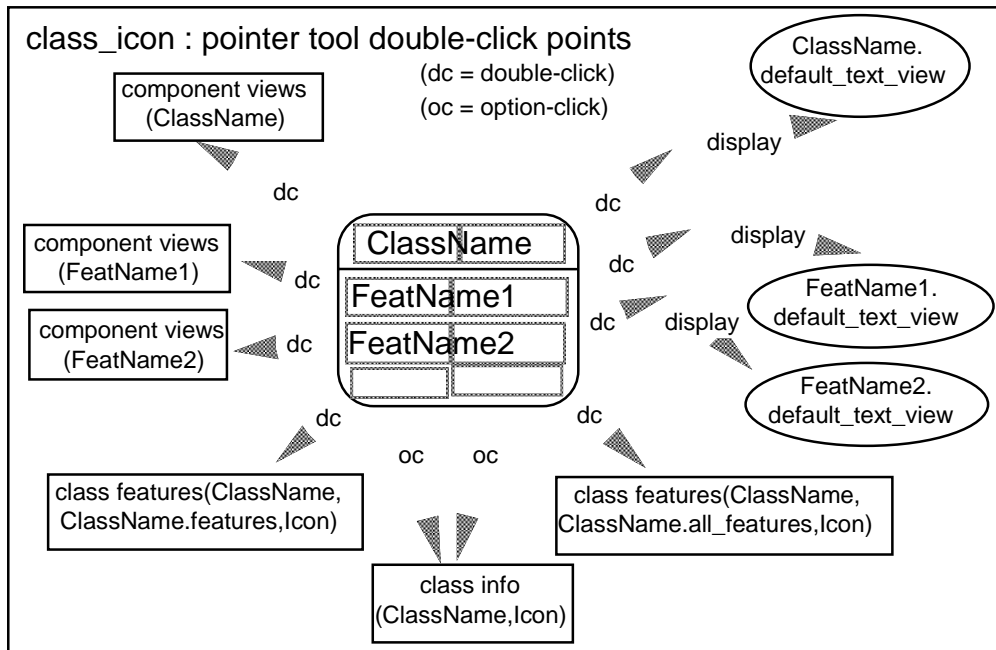
fig. 5.26. The effects of double-click/option-click actions on class icons.

5.6.4. Views

Fig. 5.27. describes the appearance of class icon views and the effects of addition tools on class diagram components. For example, the class icon tool (⬚) produces an `add_icon` update when clicked on an empty view position (this update is sent to the view itself). `add_icon` is a parameterised update which includes the kind of icon to add. Click-areas are used to restrict the application of some updates (for example, adding a feature to a class icon). Rubber-banding from one class icon to another (for example, from class icon Name3 to Name2 in fig. 5.27.) will add a generalisation, client-supplier or classifier glue connection, depending on which relationship tool is currently selected. The event to generate depending on the selected relationship tool is determined by an annotation on each of the event flow arrows from the `rubber_band` line (for generalisation, for client-supplier, and for classification). For example, if the generalisation tool () is selected, a rubber band between two class icons will generate an `add_glue(gen,Name3,Name2)` update event.

When specifying an environment using MVisual, some assumptions could be made about the graphical user interface and view support. For example, MViews and IspelM assume a Macintosh-like tool and graphics window and the provision of generic editing operations, tools and updates. For example, the selection tool (➤) and `rubber_band` editing and update behaviour is assumed to be understood (but could be explicitly defined in MVisual).
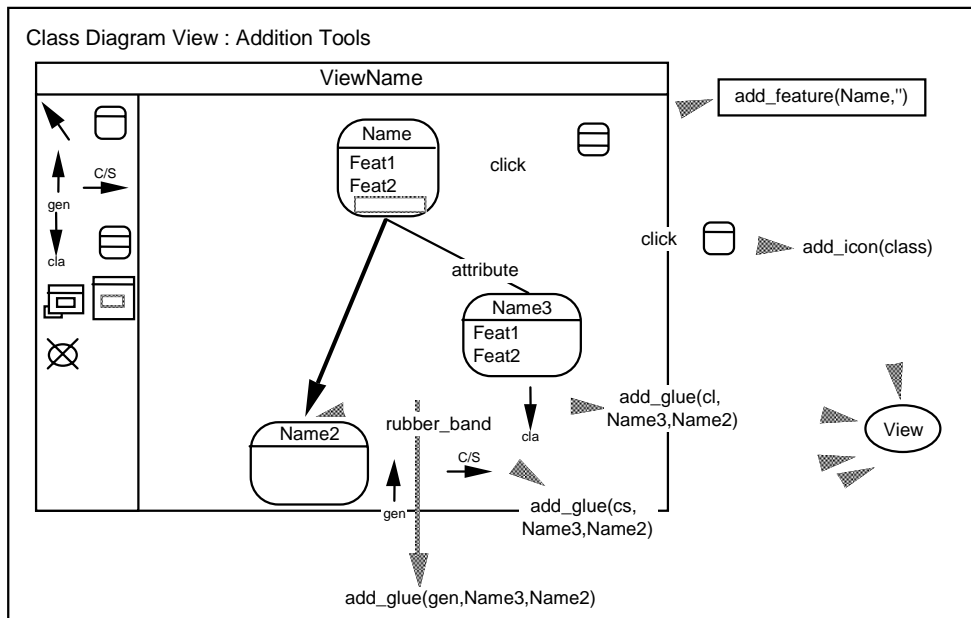
fig. 5.27. Class diagram view appearance and addition tool updates.

MViews textual view appearance and interaction is assumed to be in the form of a text window with standard text editing operations. Parsing and unparsing, however, are defined for each environment, as is update unparsing. Fig. 5.28. illustrates the application of updates to a textual view with a Snart-like syntax.
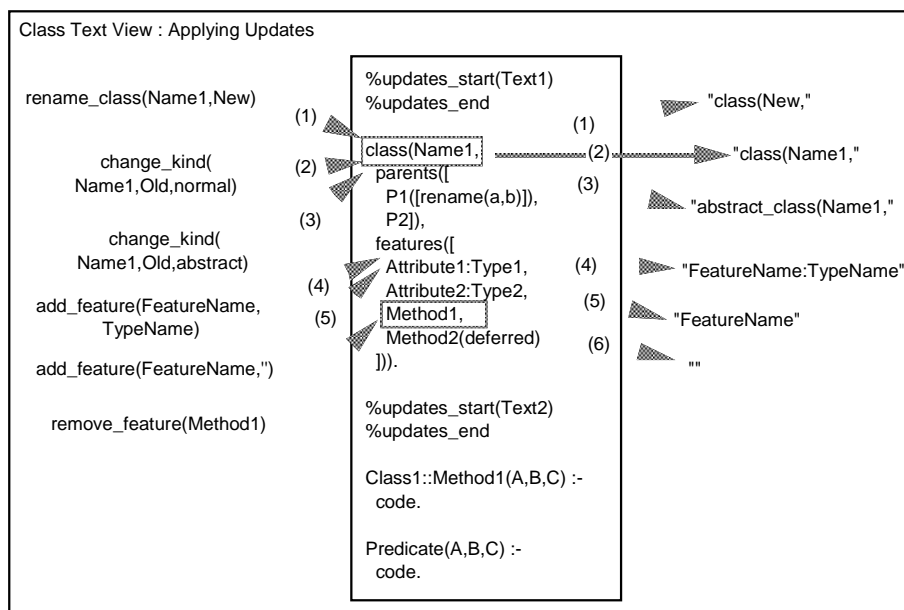
Fig. 5.28. Parsing and update application to a text view with a Snart-like syntax.

This specification uses a combination of example-based programming and visual programming to describe the effects of applying different updates to a text view. An example of textual view component renderings provides an illustration of how view components are rendered in the textual view (example-based programming). Update flows describe where update records are applied to the view's text and describe the resulting text after applying the update (visual programming). These update flows are

annotated with numeric labels to indicate which resulting text corresponds to which input update record. For example, a `rename_class(Name1,New)` update record will update text of the form "`class(Name1`", and change it to "`class(New,`". MVSL does not define the textual appearance of base elements and relationships but assumes MVisual will provide an example-based specification for rendering subset components in textual views.

5.6.5. Dialogues

Dialogues are used for structured user interaction and are specified in a similar manner to icons, glue and views. Fig. 5.29. shows the dialogue fundamentals used by MVSL. Dialogue components can be highlighted with borders to indicate groupings of values. These dialogue components correspond to those used by Macintosh dialogues but could be changed to reflect another user interface system's standard.
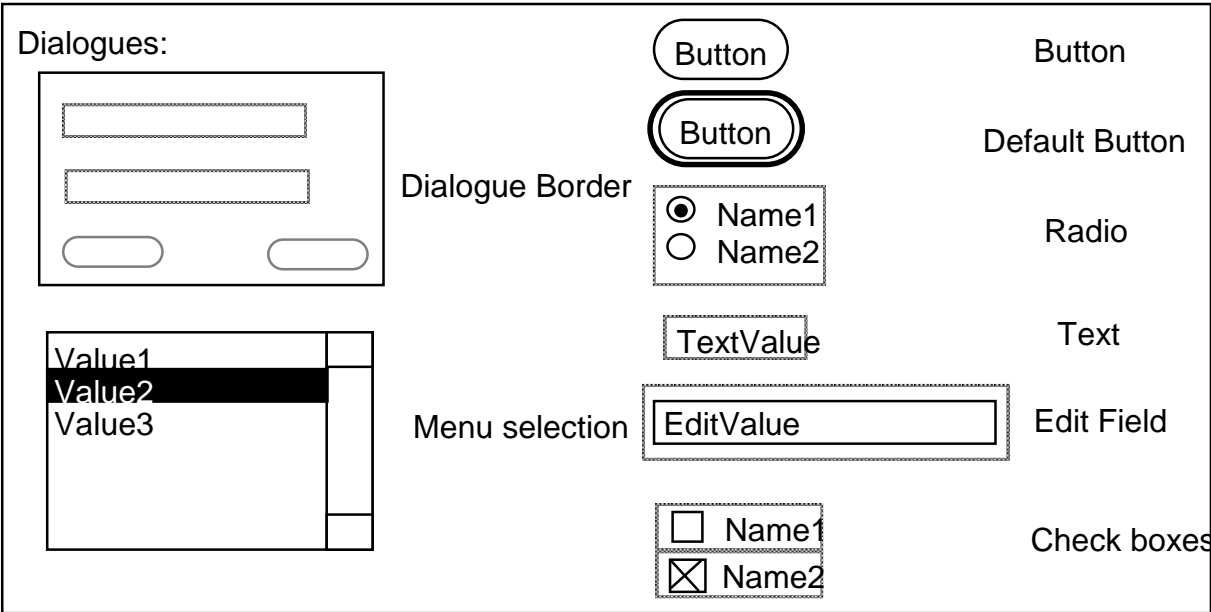
fig. 5.29. MVisual dialogue fundamental appearance and interaction components.

A simple example of MVisual dialogue specification is shown in Fig. 5.30. The effect of selecting a view name for a component in the view selection dialogue from MViews is to send a display update to the appropriate view.
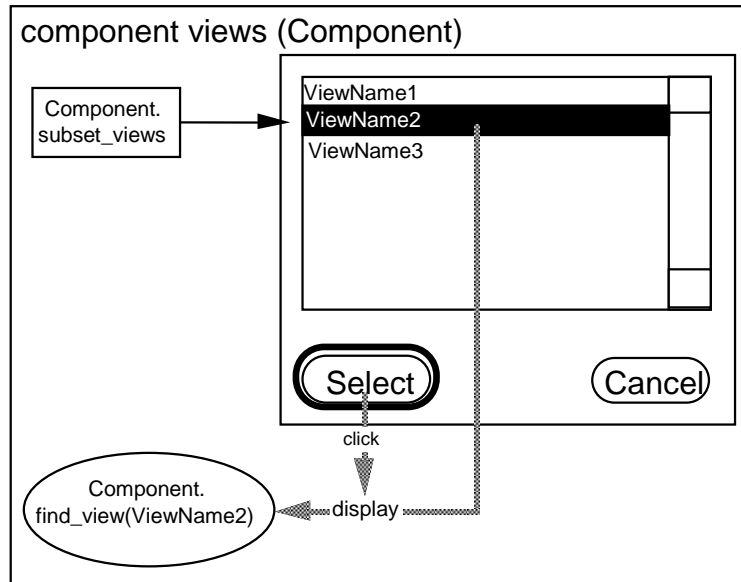
fig. 5.30. The view selection dialogue from MViews.

Fig. 5.31. shows the feature addition dialogue for IspelM. This allows programmers to add or update feature names for a class icon and also modify attributes associated with a feature.
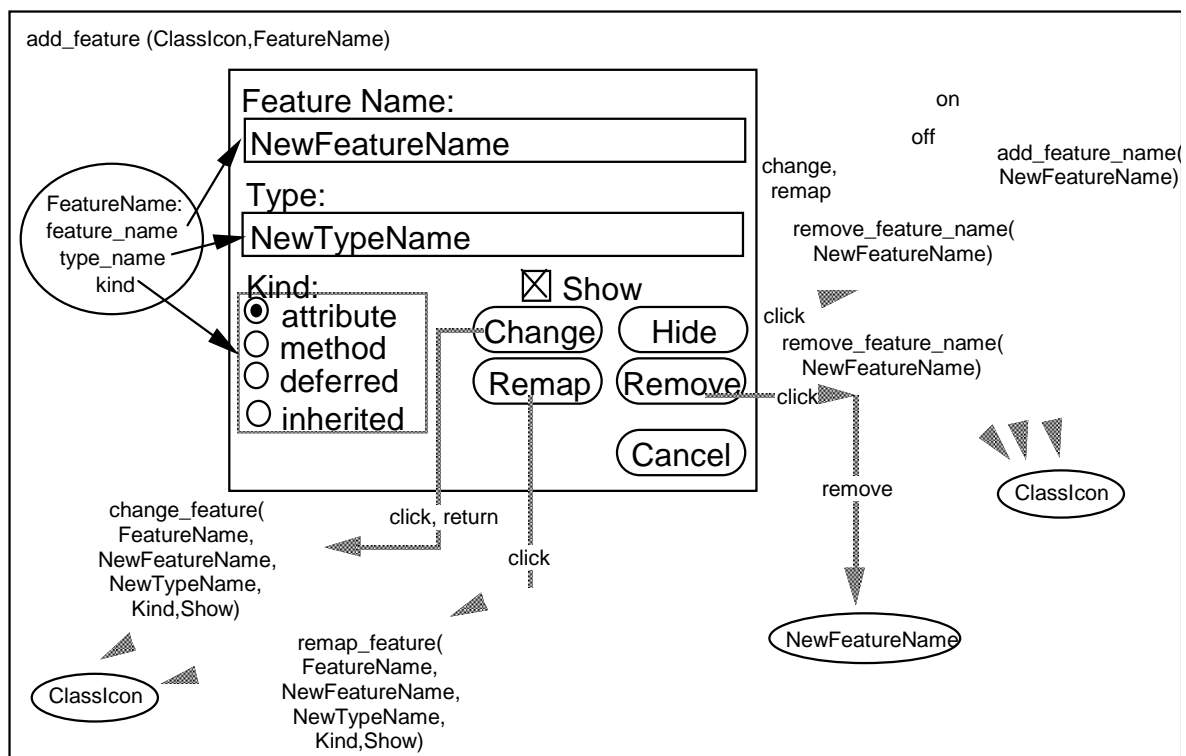
fig. 5.31. The feature addition dialogue for IspelM.

The values of `FeatureName` and `Type` are supplied by an MVSL base feature (look-up of the base feature is assumed to be implicit given a class icon/feature name pair, but could be explicitly defined as `ClassIcon.base.find_feature(FeatureName)`). `Change`, `Remap`, `Show` and `Hide` send updates to the MVSL class icon with appropriate information from the

dialogue. The `Kind` radios relate to the feature kind attribute (possible values `attribute`, `method`, `deferred` or `inherited`) and are grouped by a highlight box.

5.7. Discussion and Future Research

5.7.1. Requirements Satisfaction

Section 5.1. identified several requirements for a system for building programming environments that support multiple textual and graphical views. Section 5.2. indicated that most existing environment implementation models do not satisfy all of these requirements or provide less than ideal approaches. This section illustrates how the MViews model of Section 5.3. satisfies these requirements and proposes enhancements to MViews that could provide better support for such environments.

Program Representation

Environments require abstract and flexible program structure and semantics representation schemes. MViews supports a general model of program representation using elements, relationships and views. Program components are linked via relationships (which can represent both structural and semantic links) and all components can hold tagged attribute values. Operations can be associated with each kind of component to support component-specific manipulations.

This program representation scheme is sufficient for both high-level and low-level program components using a model of programs as graphs with nodes (elements) and labelled edges (relationships) (Arefi et al 90). It also supports graphical (visual) programming language representation which often have a graph-based structure, unlike standard abstract syntax trees (Backlund et al 90, Golin and Reiss 90). This scheme is more suitable for program representation than those of Unidraw, Dannenburg's `ItemList`, and Wilk's object dependency graphs, all of which assume components with a less appropriate structure (such as item lists, objects with references, or hierarchical graphical diagram components). Its level of abstraction can be compared favourably with abstract syntax grammars (Reps and Teitelbaum 87), decorated abstract syntax grammars (Backlund et al 90), and deterministic, directed graphs (Arefi et al 90), in terms of structural representation and manipulation (as MViews supports a basic set of graph operations similar to those of abstract syntax tree manipulation operations).

The storage of semantic attributes and relationships using the same scheme as structural components is similar to that of GARDEN. The specification of how semantic values are calculated, however, does require more effort than attribute grammars (Reps and Teitelbaum 87, Magnusson et al 90) and graph-based attribute grammars (Hudson 90, Backlund et al 90) as MViews requires recalculation to be explicitly defined using program

update detection or extra component-specific operations. For IspelM the current MViews approach has not proved too cumbersome. If the low-level static semantics of statements were to be defined (as opposed to higher-level constraints and semantics such as class interfaces), however, an attribute grammar or action equation-like (Kaiser 85) specification would be much more suitable and abstract. Such a semantics model could be used in conjunction with the current MViews representation by using program updates to drive the recalculation algorithm (as in Mjølner environments (Minör 90)) and by storing recalculated semantic values as attributes and relationships.

Multiple Textual and Graphical Views

SPE-like environments require support for textual and graphical views, preferably utilising models similar to those employed for program representation. In MViews, views are represented as graphs in the same manner as the base program graphs they view. This has the advantage that the view representation and manipulation strategy is the same as the base program and hence simplifies the specification of view structures and operations (as these can often mirror those of the base).

MViews defines subset views to be graphs which are a partial copy of the base program graph. Display views render subsets in a textual or graphical form which programmers see and interact with. This model supports textual and graphical representations of subsets of a base program and these views share the techniques used to model and manipulate the base program. Subsets can have the same structure as the base program graph or a different structure, depending on the requirements of the display/subset view. This approach achieves similar levels of abstraction to that of Unidraw (Vlissides 90) and the `ItemList` structure (Dannenburg 91).

Program and View Modification

Changes to program views must be translated to base program updates and views should supply editing mechanisms appropriate for their rendering. MViews base and subset views and components are both manipulated using graph operations. This facilitates translation of updates between the two levels as the structures updated and operations applied at one level are often very similar (or the same) as the structures used and operations required to reflect the change in the other. MViews provides a subset view and base representation scheme of similar abstraction to that of Dannenburg's `ItemList`. These are generally more efficient at translating subset and base changes than Smalltalk and Unidraw-like systems (Wilk 91).

MViews provides free-editing text views and interactively edited graphical views. Both editing mechanisms are translated into structure-editing operations but this process is hidden from programmers. In general programmers prefer to free-edit textual expressions

and control structures (as their conception of this level of a program is as tokens rather than trees and graphs) and structure-edit interfaces and inter-module (and class) relationships (Welsh et al 91). As programmers tend to find template and menu-driven structure-oriented editors cumbersome and unnatural to use (Minör 90, Welsh et al 91) MViews tries to provide the most appropriate representation for programmers given the level of program abstraction being manipulated.

One consequence of this approach is that parsing of textual views requires structures that can be compared to base components. Thus textual views can not be used to provide multiple views of control structures as these can not usually be uniquely identified. Such views are not very useful in general and MViews assumes that either the base components for such detailed views are completely regenerated by textual parsing or such detail is stored as a stream of text and given to existing compilers (with parsing only updating higher level structures such as variable declarations and class and method interfaces). For graphical languages or graphical representations of structure or semantics this problem does not occur. All icons and glue are linked directly to corresponding base components and hence can be multiply viewed and updated.

Textual structure-oriented editing of class interfaces may, for some programmers, be more appropriate than free-editing. MViews could be extended to provide textual structure-oriented editors like Mjølner or UQ2 (Welsh et al 91) by translating abstract syntax tree manipulations into graph operations with graphical views used to display an unparsed abstract syntax tree (stored as an MViews subset program graph). Generation of structure-oriented editors from grammars is usually easier and more abstract than building interactive editors and parsers by specialisation of an MViews-like model (Garlan 86, Backlund et al 90, Minör 90, Whittle et al 92). This means the construction of MViews-based environments would require more effort than comparable structure-oriented environments like Dora and Mjølner, but MViews environments may be more "programmer friendly" (as flexible support for interactive editing is provided).

Automatic, Efficient Consistency Management

Changes to base components must be propagated to all views affected by the change. Language semantics must be recalculated appropriately and incremental semantic and view updates should be made where possible. MViews uses a concept of object dependency to propagate update records describing component changes. These update records are interpreted by the dependents of a component (including its subset view components) which take appropriate action (recalculate semantic values, apply operations to update their own state, re-render their display, etc.). This propagation mechanism uses relationships and updates are automatically sent to dependents. Lazy application of updates could be used for views and semantic recalculation for efficiency (Wilk 91).

MViews supports incremental view updates as the exact change is sent to subset view components. These can make changes based on the precise change to their base component state and display views need only redisplay information based on updated subset components. An added advantage of MViews over Smalltalk and Unidraw-like models is that subset view components do not have to repeat base component information. For example, textual view components can expand the base updates sent to them to indicate the change made whereas the other models must repeat the base information to be able to determine the kind of change made. If subsets do, in fact, duplicate base information, the exact change to subset components can be determined from base update records.

MViews does not directly support the notion of object inferiors, superiors and transitive object dependencies, as supported by (Wilk 91). Currently, if an object X is dependent on an object Y, X is always informed of changes to Y but only informed of changes to Y's component objects (part-of relationships) if Y decides to broadcast its sub-component changes. One approach to enhancing this support might be to introduce explicit part-of relationships which would automatically broadcast changes to sub-components of a component to the component's dependents.

Recording Previous Changes

Providing a change history for a program component is useful for documenting the modification history of program components. MViews supports such a facility by allowing update records to be stored against program components for future reference. Such a facility is not directly supported by most other models which may not allow such a facility to be easily implemented. For example, the Smalltalk MVC model can not always determine the exact change made to a model and Dannenburg's ItemList and Wilk's object dependency graphs do not allow recorded changes to be directly accessed.

Undo and redo of User Manipulations

Undo and redo of user manipulations is necessary so programmers can undo (or redo) changes they have made which may have had unseen transitive effects (since the updates will have been propagated to affected base components). MViews supports a generic undo/redo mechanism by recording update records generated by subset view components. These can be sent back to the components for reversal or reapplication. Such a facility can also provide a transaction mechanism where all program state changes for an editing operation have to be reversed if the operation is invalid (i.e. the operation is aborted). MViews provides a facility of comparable abstraction to GARDEN and Unidraw but it is also compatible with semantic recalculation and view-to-base consistency management (undoing an update record generates operations which in turn generate further updates etc.).

MViews could be extended so that updates could be undone or reapplied out of strict sequence order. This would provide a mechanism for arbitrary undo/redo of view updates and perhaps "macro operations" where updates are reapplied to different view states (similar to PECAN's macros based on an undo history (Reiss 85)). The main problem with such a mechanism is that the operations performing the undo/redo must check whether the current program state will support such an action (as updates undone out of order may be inconsistent). This is discussed further in Chapters 7 and 10.

Program and View Persistency

A program must be persistent over successive invocations of an environment. While the MViews model does not assume a specific persistency mechanism, one can be modelled using operations to save and load a component's state. Version control, configuration management and multi-user access to programs are not currently supported. These could be modelled, however, by grouping updates in versions and by programmer (i.e. by who generated the update). Multiple versions could be supported by allowing programmers their own distributed workspace, similar to (Nascimento and Dollimore 93), and merging of versions by merging update records associated with different versions. Like arbitrary undo/redo, this would require testing of updates to ensure undo/redo is consistent given a program graph state (as updates may be done out-of-sequence or applied to program components that no longer exist in the current version). Chapter 10 discusses these issues in more detail.

Tool Integration and Extensibility

Environments need to provide consistent user interfaces, a common tool data storage or translation mechanism and allow new or exsisting tools to be integrated into the environment. MViews supports tool user interface integration via a consistent dialogue and display view interface. Tools use the base view for a cannonical data storage repository and define subset views which can provide a partial view of the base and even different structures to the base. Subsets can also be used to export and import data from tools and use subset components to relate external tool data to internal base view data. Update records could be used in an analagous manner to FIELD's selective broadcasting system (Reiss 90a) for propagating changes to MViews structures to and from external tools via subset views.

5.7.2. MVSL

MVSL provides a simple specification language for MViews systems. A basic set of "component kinds" is provided to capture the fundamental abstractions of MViews environments with a component's state defined by typed attributes and relationships to other components. A basic set of operations is provided for manipulating MViews

programs which can be augmented with component-specific operations. Updates are interpreted by each component using guarded input-only operations which carry out further operations in response to an appropriate update record.

MVSL is sufficient to abstractly model the basic concepts of component state, operations and update responses for MViews systems. Component relationships and response to updates are explicitly defined. This approach directly supports the object dependency model of MViews to be concisely and clearly expressed, unlike most specification languages where such facilities must be modelled with more basic structures.

MVSL is not a fully-fledged programming language and it is difficult or impossible to express some concepts. For example, there is no concept of reusable operations or functions and inheritance is not supported between components, making MVSL only suitable for abstract environment state analysis and not detailed specification. In addition, MVSL assumes all components connected to a component are dependents and propagates update records to them (perhaps unnecessarily). The scheduling of update record propagation is assumed to be immediately following the application of an operation, which does not allow for efficient implementation using lazy application of update operations. Since MVSL is strongly typed it is difficult to express the management of update records which are lists of values of arbitrary types.

As MVSL can not be used to completely specify an environment, we have developed an object-oriented architecture for MViews systems based on the concepts introduced in this chapter. This can be used as the basis for an object-oriented implementation of MViews and hence be reused to design and implement new environments. MVSL can be used as a preliminary analysis tool or documentation aid and then the object-oriented architecture used to model an environment in more detail using class specialisation. Chapter 6 describes this architecture while Chapter 7 provides a Snart implementation of this architecture. This object-oriented design and implementation allows MViews environments to be modelled and efficiently implemented by specialising a reusable framework of classes.

5.7.3. Operational Specification of MVSL

The operational specification of MVSL illustrates that an MVSL program is well-defined in terms of its effect on an MViews program state. An MViews program is stored as a `State` containing component attribute values which represents an instance of the MVSL program specification. Meaning functions for expressions, commands, basic operations, component-specific operations and update operations are defined which return a new `State` given an MVSL construct. The meaning of an MVSL program is a list of update records generated by MVSL when "executing" an environment specification given a list of input update records.

This specification can be improved by defining the static semantics of an MVSL program. Currently an MVSL program is not type-checked before execution and errors are indicated by Gofer function exceptions in the output update stream. To support functional operations with side-effects (i.e. that can change the program state) the result of exp_val should be a new `State` as well as a denotable value. A continuation-style specification could be used to produce error messages when invalid operations are attempted.

5.7.4. MVisual

MVisual allows the user interaction aspect of MViews environments to be defined using a natural graphical specification "language". MVisual provides a graphical notation for specifying the appearance of MVSL subset views, elements and relationships. It also provides a mechanism for specifying how users interact with these visual entities and the effects of such user interaction (in terms of update record flow). MVisual uses MVSL component values to define where visual entity values are derived from and passes updates to MVSL components as a result of visual entity modification. The MVisual notation is less rigorous than MVSL with assumptions being made about editing tools and dialogue appearance and behaviour and permits partial specification of component behaviour.

MVSL and MVisual interact using update records. MVisual assumes MVSL will interpret update records sent to components appropriately and MVSL assumes MVisual will interpret the updates it generates. This model assumes each notation will synchronise input and output appropriately and the operational specification for MVSL assumes this. While MVisual provides a concise, natural mechanism for expressing the appearance of views and dialogues, and user interaction with these entities, it is not ideal for all such specification. Particular failings are when trying to specify constraints on dialogue interaction, complex MVisual to MVSL to MVisual update flow, and explicit requests for user input from MVSL.

For dialogue constraints, the values of different edit fields may depend on values of radios and other edit fields. For example, in the client-supplier update details dialogue `Client Name` and `Supplier Feature` values are only valid for certain values of `Kind`. Expressing these constraints with MVisual notation becomes quite cumbersome, especially when error actions or edit field skipping are to be defined (i.e. error message reporting and/or specifying that an edit field is not to be used given certain constraints). A textual specification of such constraints and error message generation may be more concise than a graphical one.

When a complex flow of control from MVisual to MVSL and back to MVisual occurs, MVisual does not clearly indicate that the flow back from MVSL is a result of the original user interaction. For example, an `expand` update sent to a subset view from a dialogue

could use an `add_view_component` update from MVSL to indicate an expansion into the display view is required. Currently, such an expansion for IspelM is specified as a response to the original MVisual `expand` update in a different MVisual view with no indication that MVSL performs `add_element` and `add_view_component` operations. An indication of such flow-of-control would be useful for clarity in MVisual.

Both MVSL and MVisual are currently used by defining an MViews environment using a drawing program and text editor. An MViews-like environment supporting multiple views of an MVSL/MVisual specification would be make this definition process much easier and allow a specification to be browsed. It may also permit limited generation of reusable classes from the architecture in Chapter 6 from MVSL/MVisual specifications. Chapter 9 briefly discusses the requirements for such a specification environment.

5.8. Summary

SPE-like environments require a flexible program representation scheme, support for modelling multiple textual and graphical views of programs, and editing operations to manipulate these representations and views. They also require efficient, automatic detection and propagation of changes to support view consistency and language-specific semantic recalculation. View editing should be appropriate for the view's rendering, a generic undo/redo facility should be supplied by the environment, and an abstract program saving and reloading mechanism be supported.

MViews provides a novel set of abstractions for implementing such environments based on object dependency graphs. Programs are represented as graphs made up of elements and relationships grouped by a base view. This representation is sufficient for storing structural and semantic information for both tree-based and graphical languages. Views of this program graph are represented in the same manner by subset view graphs and these subset views are manipulated using the same graph operations as the base program graph. Subset views can be displayed and edited as either text or graphics. Update records are generated to document component changes and these are propagated to dependent components. Update records can be used to translate changes between subset and base view components (and vice-versa), be recorded to document changes to components, used to implement a generic undo/redo facility, provide incremental, efficient subset/display view updates, and drive semantic recalculation.

MVSL is an abstract specification language used to describe the state of base and subset graphs, and the editing semantics of these graphs, for an MViews environment. An operational specification for MVSL illustrates that the basic concepts of MViews can be captured using an object dependency graph state and basic graph manipulation operations on this state. MVisual provides a mechanism for specifying the display view and user interaction component of an MViews environment. MVisual utilises example-

based and visual programming-based specification techniques to describe the appearance, effects of user interaction and effect of MVSL operations on the user interface for an environment. MVisual and MVSL interact using update records to pass changes between the subset and display levels of MViews.

Neither MVSL nor MVisual can currently be used to specify enough information for an environment implementation. Chapter 6 demonstrates how the basic abstractions of MViews can be used as the basis for an object-oriented architecture for designing an implementation of an environment. This architecture is comprised of classes which are specialised to describe environments like IspelM. Chapter 7 shows how a Snart implementation for MViews can be derived from this architecture. Chapter 8 uses the architecture of Chapter 6 to produce a model for IspelM and uses the Snart framework of Chapter 7 to implement this model. This implementation of IspelM is then further specialised to produce an implementation for SPE.

Chapter 6

An Object-Oriented Architecture for MViews

Chapter 5 describes the MViews model for interactive software development environments. MVSL is used to abstractly specify the state and editing semantics for base and subset views of an environment using a textual language. MVisual is used to abstractly specify the display views and user interface for an environment using a graphical notation. Neither of these specification languages, however, are sufficient for deriving an implementation of an MViews environment.

This chapter describes a language-independent, object-oriented architecture for MViews. Component kinds are described by classes, component attributes by attribute classes, and operations by class methods and an MViews environment program is stored as objects (instances of these classes). A new environment is constructed by specialising this framework of classes appropriately. This object-oriented design for MViews is much more suitable for implementing an environment as it provides more detail than MSVL and MVisual and is much closer to an (object-oriented) implementation language. Since this architecture is derived from the fundamentals of Chapter 5 it can be used to translate MVSL and MVisual specifications for an environment into an object-oriented design. This design can then form the basis for an implementation of the environment.

The rationale for an object-oriented architecture for MViews is discussed and an overview of the fundamental classes for the architecture is given. Each group of related classes is then described in more detail with the purposes of their major attributes, methods and interactions with other classes explained. Chapter 7 uses this object-oriented architecture for MViews as the basis of an object-oriented implementation of MViews as a framework of Snart classes. Chapter 8 uses this architecture and the Snart framework of Chapter 7 to model and implement IspelM and SPE.

6.1. An Object-Oriented Architecture for MViews

As discussed in Chapter 2, several approaches to implementing programming environments are possible. To produce a reusable MViews system either a programming

environment (PE) generator with its own specification language (similar to MVSL) could be implemented, or a specialisable framework of classes used. The Synthesizer Generator (Reps and Teitelbaum 87), MELD (Kaiser and Garlan 87), and Mjølner/ORM (Magnusson et al 90) provide specification languages based on abstract syntax and attribute grammars which are translated into an implementation. Unidraw (Vlissides 90), (Haarslev and Möller 90), and Interviews (Linton et al 88) provide object-oriented frameworks for implementing drawing editors, visualising object-oriented systems, and constructing graphical user interfaces respectively. We chose the second approach for several reasons:

- Many aspects of a good, interactive PE, such as editor functionality and interfaces, require specialisation and fine-tuning on a scale difficult to provide with a specialised PE generator (Vlissides 90, Ratcliffe et al 92). A reusable, object-oriented framework allows more flexible extensions to be implemented and reuse of existing code libraries and tools. It provides the full power of a general-purpose programming language but within a conceptual model (the reusable framework) for the environment.

- Generated PEs often provide poor or inappropriate user interfaces and can lack adequate response-time performance (Minör 90). The most common approach to generated environments involves producing structure-oriented editors from abstract syntax descriptions (Reps and Teitelbaum 87, Minör 90, Ratcliffe et al 92). Structure-oriented editing of both text and graphics is a common feature of such languages (Whittle et al 92) but this approach has yet to gain wide-spread favour with programmers (Minör 90, Whittle et al 92, Welsh et al 91). Reusable frameworks can provide a more tailorable, interactive model of user interaction (Vlissides 90).

- Generated environments provide a high-level of abstraction in both the specification of their program structures and semantics and their editing operations (Minör 90, Reps and Teitelbaum 87). Disadvantages with this level of abstraction, however, are the implicit constraints put on environment implementers with respect to adding flexible language semantics (Kaiser 85, Hudson 90) and lack of general-purpose programming power for implementing unusual or extended facilities (Kaiser and Garlan 87, Vlissides 90). An object-oriented framework can achieve a reasonably abstract representation of a conceptual model via good use of appropriate abstractions (Vlissides 90) while still incorporating flexible, general-purpose programming facilities.

- As we wanted to experiment with parts of the MViews model during development to determine appropriate approaches, we did not initially know what a PE generator language for MViews should support. An extensible object-oriented framework supported a more flexible, experimental development platform for modelling MViews environments.

A partially-generated MViews environment may provide a good compromise between the desire for a concise, abstract specification of an environment and the requirement of a flexible, efficient implementation to provide a useable end-product. Chapters 9 and 10 briefly discuss using MVSL and MVisual to generate classes specialised from classes in the MViews architecture. These classes could then be further specialised to implement efficient or unusual language structures, semantics and editing tools.

6.2. Overview of the MViews Architecture

6.2.1. Components as Classes

The MViews architecture defines classes based on the abstractions described in Chapter 5. Extra abstractions are introduced to allow more precise modelling of different environment facilities and to make the architecture more reusable (by providing extra reusable components and additional functionality). Fig. 6.1. shows the hierarchy of classes for MViews. MVSL basic component kinds are modelled as classes (for example, base views as `base_view` and graphical icons as `graphic_icon`). MVSL component attributes are represented by objects associated with a `component` and relationships are defined as attributes which refer to relationship component objects.

Environment-specific component kinds are defined by specialising classes appropriately. For example, a `class_icon` for IspelM can be defined by specialising `graphic_icon` from the MViews framework and defining appropriate extra attributes (such as `class_name` and `feature_names`) and methods (`update_attribute` for `class_name` and `add_feature_name` for `feature_names`).
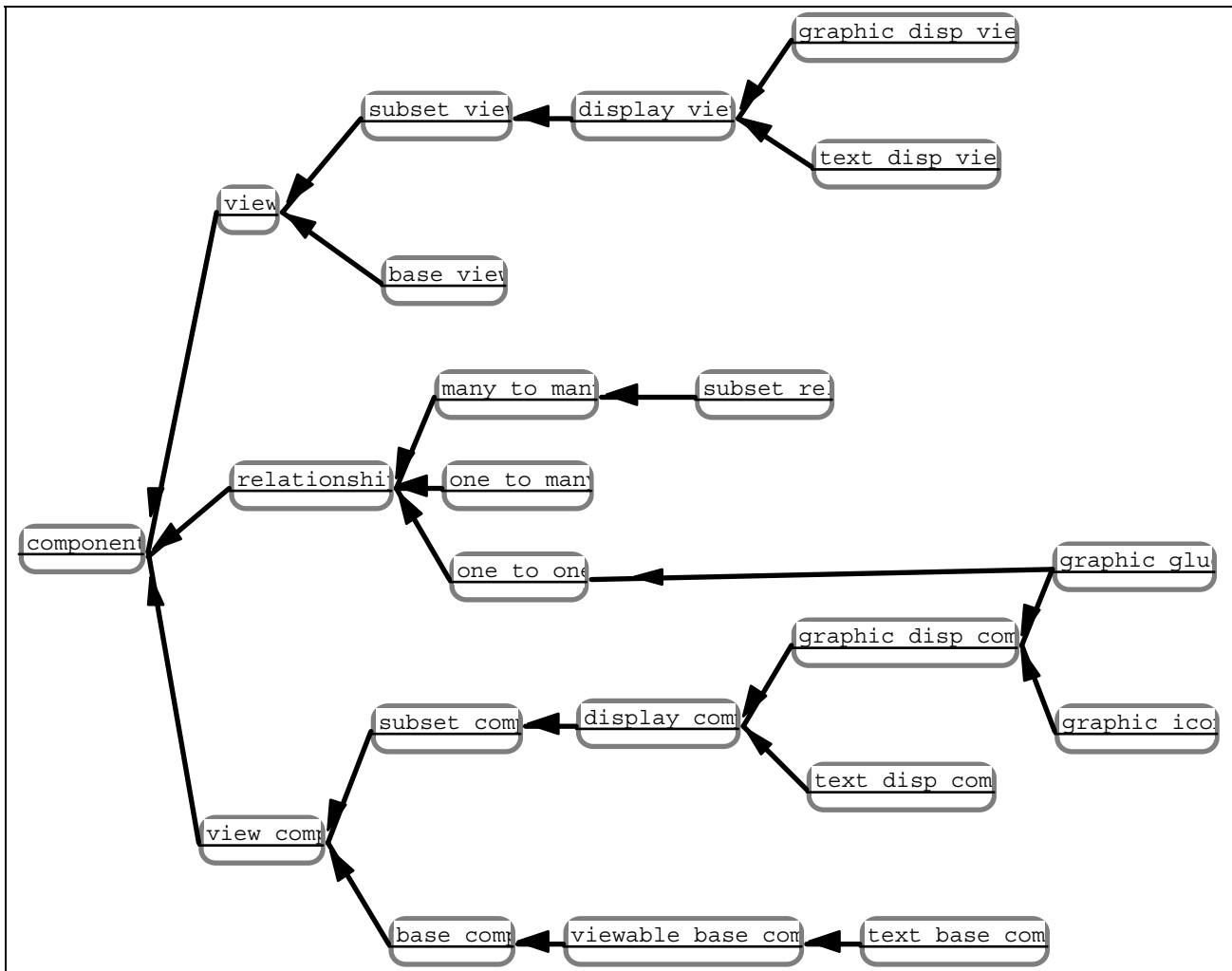
fig. 6.1.   An object-oriented hierarchy of MViews components.

Table 6.1. illustrates how the MVSL and MVisual components are mapped onto MViews
architecture classes. An MVSL component is implemented by a class specialised from one
or more MViews architecture classes. An MVisual component is implemented with its
corresponding MVSL component as a class specialised from one or more MViews
architecture classes. Additional, abstract classes are introduced by the MViews
architecture to factor out common data and behaviour from different MViews
components. For example, `component`, `view` and `relationship` classes do not have direct
MVSL equivalents but are used to capture common component, view and relationship
behaviour. New environments do not use these abstract classes directly but specialise new
classes from those shown in table 6.1. The following sections briefly describe how MVSL
and MVisual components are implemented by these architecture classes and what extra
information these classes provide.

MVSL/MVisual Components	MViews Architecture Class(es)
base view	`base_view`

base element	One of:
	    `base_comp`
	    `viewable_base_comp`
	    `text_base_comp`

base relationship	One of:
	    `base_comp`
	    `viewable_base_comp`
	    `text_base_comp`
	One of:
	    `one_to_one`
	    `one_to_many`
	    `many_to_many`

subset view
display view
	One of:
	    `graphic_disp_view`
	    `text_disp_view`

subset element and graphical display icon	`graphic_icon`
subset relationship and graphical display glue	`graphic_glue`
subset component and textual display component	`text_disp_comp`
subset to base component relationships	`subset_rel`

Table 6.1. Mapping of MVSL and MVisual components onto MViews architecture classes.

6.2.2. Base Components

A base view in MVSL groups base components and base views are implemented by specialising the `base_view` class. `base_view` supplies features for locating components using unique identifiers and look-up tables, managing subset views, and mapping between components and their kinds. All components are created by calling methods supplied by `base_view`[13].

Base elements from MVSL are implemented as classes specialised from `base_comp`. Additional classes are introduced for modelling base components that can have subset components (i.e. can be viewed), as `viewable_base_comp`, and base components which can have textual forms, as `text_base_comp`. `viewable_base_comp` provides features for managing subset view components, including view management and navigation facilities. `text_base_comp` provides additional features for managing text forms associated with base components, and base components with textual view renderings are specialised from this class. Base relationships specialise `base_comp`, `viewable_base_comp` and `text_base_comp`, but also specialise one relationship class (`one_to_one`, `one_to_many` or `many_to_many`). These relationship classes provide features for representing and managing component relationships and may use different kinds of collection classes to group the components of the relationship (usually lists).

Implementing MVSL base components as classes specialised from a variety of MViews architecture classes allows environment implementers to specify more detailed base

---

[13] This can be used to assist in supporting environment evolution, as described in Section 6.7.

component functionality. The main advantage of the architecture classes is the additional data and behaviour they provide for managing views and view components, managing text forms, and managing component relationships. MVSL specifications ignore the detail of these tasks and are thus further refined when they are modelled by specialising classes from the MViews architecture.

6.2.3. Subset and Display Views and Components

MVSL describes subset components while MVisual gives the display component rendering for these subset components. While this separation is useful for the purposes of abstract specification, it is not usually useful to implement subset components and display components independently, as they are closely interdependent. A display needs partial views of base component values to render and update (i.e. a subset component state and operations) while a subset component requires a rendering which it must inform of changes to itself so it can be re-rendered (i.e. a display component). Subset views and their corresponding display views are modelled by specialising `graphic_disp_view` and `text_disp_view`. These display view classes contain the subset view management features and tools for rendering and editing graphical or textual view components.

Subset elements rendered as graphical icons are specialised from `graphic_icon`. `graphic_icon` includes features for managing a subset element and features for rendering and manipulating a graphical icon. Subset relationships rendered as graphical glue are specialised from `graphic_glue`. Subset components rendered as text forms are specialised from `text_disp_comp`. `text_disp_comp` provides features for manipulating text forms including unparsing and inserting readable update record descriptions, applying selected update records, and determining the text associated with a textual display component.

MVSL and MVisual subset and display views and components are modelled by classes specialised from these MViews architecture classes. The use of one MViews architecture class for each MVSL/MVisual subset and display component pair provides a concrete link between a view's program graph-based state and its rendering and manipulation. The MViews architecture classes also provide additional features which support view navigation, graphical and textual component manipulation, graphical and textual editing tools and an undo/redo mechanism. These allow new views and their editors to be quickly defined based on an MVSL view state specification and MVisual appearance/interaction specification for an environment.

6.2.4. Subset/base Relationships

MVSL subset components specify the base components they are mapped to by relationships. These subset/base relationships are implemented by specialising `subset_rel` (itself a many-to-many relationship). `subset_rel` defines a general view

consistency mechanism whereby subset to base component attribute mappings are supplied and `subset_rel` keeps these attributes consistent under change. Any additional view consistency functionality is expressed by specialising `subset_rel` (for example, automatically expanding components into a view when new base relationships are established). `subset_rel` also provides additional features for lazy view consistency management. This `subset_rel` relationship class allows new environments to quickly specify base and subset component attribute consistency. `viewable_base_comp` also uses `subset_rel` relationships to maintain base component to subset view relationships (used to support view navigation).

6.2.5. Additional Abstract Classes

The MViews architecture defines several additional abstract classes which are briefly described here. These abstract classes are not specialised by new environments but are used to abstract out common functionality from other MViews architecture classes.

The `component` class generalises data and behaviour common to all MViews components. These include modelling the basic operations of MVSL as methods. For example an `update_attribute(Comp,Name,NewValue)` operation is done by a method call of the form `Comp.update_attribute(Name,NewValue)`[14]. Basic operations could now be thought of as being applied to components in the same manner as component-specific operations in MVSL. All kinds of MViews components generalise to `component`.

MViews views group program graphs and the `view` class provides methods for managing view components. This includes `add_view_component` and `remove_view_component` methods (for the corresponding MVSL operations) but also additional methods for iterating through these view components and deleting the components when the view is deleted.

Relationships are modelled by the `relationship` class which provides methods for establishing, reestablishing and dissolving relationships. Relationship arity can be one-to-one (i.e. relates a parent and child), which is modelled by `one_to_one`, one-to-many (relates one parent to many children), modelled by `one_to_many`, or many-to-many (relates many parents to many children), modelled by `many_to_many`.

`subset_view` models the subset view state and operations for MViews. `display_view` is a specialisation of `subset_view` and models display view-related concepts such as tools, menus and interactive editing of display view components. A display view can render a

---

[14]We use an Eiffel-like syntax for describing method calls and attribute access as "feature calls". This syntax is the same as that used by MVSL for attribute and relationship access and operation application.

subset view either textually or graphically and hence `graphic_disp_view` and `textual_disp_view` specialisations of `display_view` are introduced. These provide extra state and operations specific to each kind of rendering and view editing mechanism.

The MViews architecture defines `subset_comp` to describe subset component state and operation semantics. The display component rendering and editing semantics is defined by `display_comp`, a specialisation of `subset_comp`. Thus the MViews architecture captures the rendering and editing semantics for a display component in `display_comp`, which also includes subset component structure and semantics inherited from `subset_comp`. `display_comp` and its specialisations provide additional methods for display component rendering and manipulation.

6.2.6. User Interface and Persistency

MViews assumes a language-specific user interface which provides menus and dialogues for display views and display components. Component persistency is assumed to be a language-specific issue which may be implemented by explicit `save` and `load` component methods or an invisible object persistency or object-oriented database mechanism. Chapter 7 describes a user interface and persistency mechanism for the Snart framework implemented by Snart methods and LPA MacProlog.

6.2.7. Summary

This overview illustrates how MVSL and MVisual specifications can be modelled using classes which are specialised from MViews architecture classes. Attributes and relationships are modelled as class attributes and class attributes with relationship components respectively. Fundamental MVSL operations, component-specific operations and update operations are modelled as class methods. The MViews architecture classes provide many more data and behavioural abstractions than MVSL and MVisual specifications. These extra abstractions allow MVSL and MVisual specifications to be further refined to include view management for base components, rendering and editing for display components, a concrete link between a view's state (subset view) and rendering/interaction (display view), and many more facilities. In the following sections the MViews architecture class structures and methods are explained in more detail.

6.3. Components

Fig. 6.2. shows the basic structure and methods provided by the `component` class using the class diagram notation from SPE.

6.3.1. Component State

`component` defines the state of all MViews components and provides methods for manipulating this state. Attributes are stored as a one-to-many relationship `attributes` to `attribute`. Each `attribute` has a `value` (which is one of `str_value` (strings), `int_value` (integers), etc. as appropriate). Attributes are accessed via `get_attribute` and modified via `update_attribute`. These attribute manipulation methods could be augmented for specialised kinds of attributes, for example, handling list attributes. In addition, while some attributes may hold values, others can compute values (for example, all ancestor classes for a class in IspelM). MViews supports data-driven programming by extending these operations or changing the behaviour of `record_update` and `update_from` (see Chapter 7 for a more thorough discussion of this).
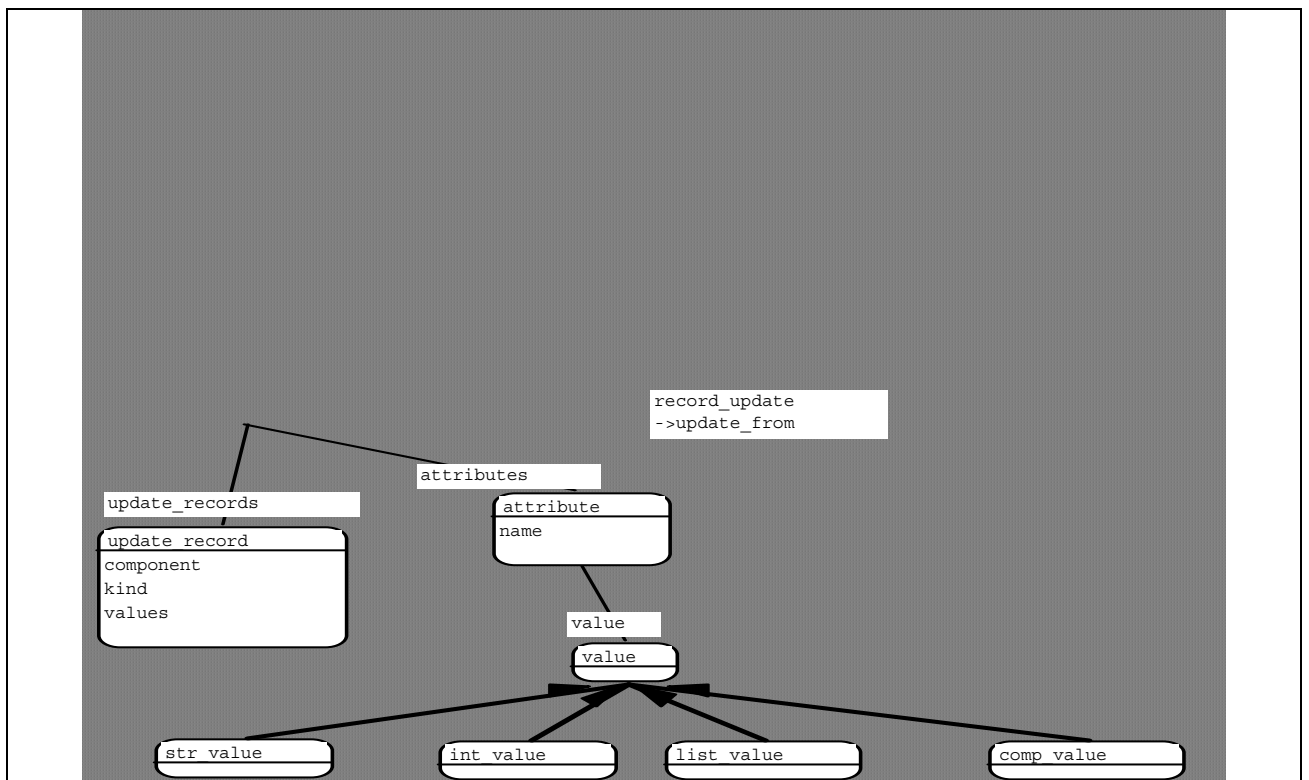
fig. 6.2.   Basic structure and methods for `component`.

Relationships may be established and dissolved by `establish_rel` and `dissolve_rel`. As operations are applied to components via method calls, an MVSL operation `establish(Kind, Parent, Child, NewRel)` equates to either of the method calls `Parent.establish_rel(Kind, Parent, Child, NewRel)` or `Child.establish_rel(Kind, Parent, Child, NewRel)` (depending on whether the relationship is to be established by the parent or child component).

The component kind for a specialisation of component can be determined by the method call `comp_kind`, which returns a string equating to an equivalent MVSL component kind

(i.e. the call `ClassIcon.comp_kind(Value)` for some `class_icon` instance will return `Value` = "class_icon").

6.3.2. Update Records

Update records are generated by operations modifying a component state. For example, the method call `Parent@establish_rel(Kind, Parent, Child, NewRel)` will generate the an update record with kind = `establish` and values `NewRel, Parent, Child`[15]. A shorthand notation used in this chapter for update records is `establish(NewRel, Parent, Child)`. This is a "term" with functor the `kind` of the update record, first argument the `component` the update record was generated by and remaining arguments the `values` held by the update record.

Update records are propagated for a `component` by calling `record_update(UpdateRecord,UpdateName)`. `update_dependents` returns a list of dependent component objects for a component and `record_update` sends updates to these dependents by calling `update_from(UpdateRecord,UpdatedComponent)`. Dependents implement an `update_from` method which decodes updates and takes appropriate action (for example, applying further updates to the dependent by calling methods). `store_update(UpdateRecord)` is used to store an update record against a component (i.e. to document the changes the component has undergone) and stored update records are maintained by a one-to-many relationship `update_records` to `update_record`.

Various operations are employed to maintain a list of a component's dependents (`add_dependent` and `remove_dependent`) and `update_dependents` can be over-ridden in sub-classes of `component` to define certain relationships to always relate dependents to a component. When a component generates an update record by calling `record_update`, `set_updated` is called which indicates a component's state has been changed (by setting the `updated` flag to true). This can be used for persistency management and for determining that attribute recalculation should take place for the component.

6.3.3. Persistency

Components are assumed to be made persistent and reloaded from persistent storage in an implementation-language dependent manner. A component can be deleted, however, which means its persistent form (and those of any of its sub-components via part-of relationships) would be removed. Thus the `delete` method equates to the MVSL `delete_component` operation.

---

[15]The "Kind" value does not need to be stored in the update record as this can be determined from the component kind of `NewRel`.

6.4. Base Program Components

Base components are used to represent the canonical form of a program. They equate to the dictionary information of CASE tools such as the OOATool (Coad and Yourdon 91) and TurboCASE (StructSoft 92) and database representations of Dora (Ratcliffe et al 92) and FIELD (Reiss 90b). Base components can store update records to document changes to they have undergone. Fig. 6.3. shows the structure and methods for each kind of base component and associated classes.

6.4.1. View Components

The `view_comp` class defines `view` to be the view a component can be contained in (a base view or subset view). MVSL assumes components define this as a relationship but our architecture allows an object attribute to be used for this purpose. View components define a name they are referred to by their owning view and `view_name` returns this as a string (for example, an IspelM base class might return "Base Class ClassName" where `ClassName` is equivalent to the MVSL value for `class.class_name`). View components also define a user name (as a string returned by `user_name`) which corresponds to the component kind (`comp_kind`) for a component but in a printable form (for example the user name for `class_icon` might be "Class Icon").
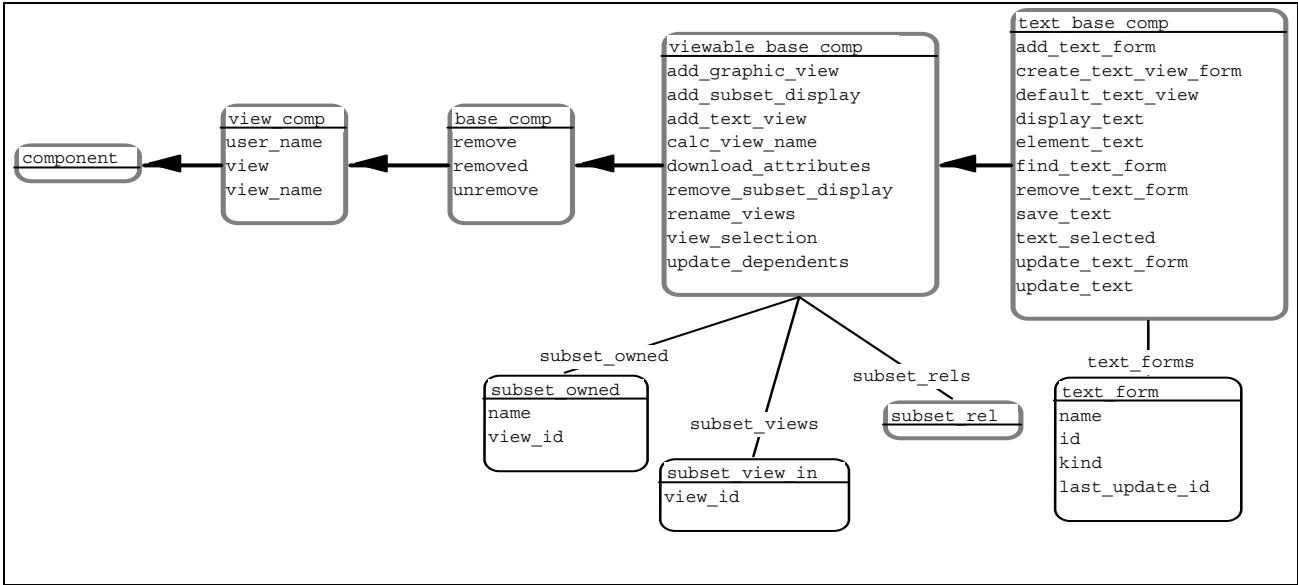
fig. 6.3.   Base components structure and methods.

6.4.2. Base Components

Base components are marked for removal using `remove`, rather than directly calling their `delete` method. This allows such operations to be reversed with `unremove`, and also allows MViews to "garbage-collect" all removed components together, limiting the effect on interactive performance.

IspelM defines `base_cluster` as a specialisation of `base_comp` (see Chapter 8 for further details).

6.4.3. Viewable Base Components

Some base components can be rendered in views (for example, IspelM generalisations and classifiers), some can have text forms of themselves (classes and features), while some can not be viewed at all (clusters). `viewable_base_comp` provides attributes and methods for maintaining views of a base component.

Subset component references can be added to and removed from a base component (using `add_subset_display` and `remove_subset_display`) and this establishes a subset/base relationship (`subset_rel`) between a base component and its subset components. The views a base component's subset components are contained in are stored against the base component in `subset_views` (as a reference to the view, possibly a unique id). This allows a base component to provide a view selection dialogue for navigating through its views (by calling `view_selection`). Views that a base component owns (i.e. is the focus of) are stored in `subset_owned` as these views must be deleted if the base component is deleted. Views a base component owns need to be renamed if base component attributes used to construct the view name are updated. `rename_views` propagates this rename to all owned views of the base component .

Base components can be created and initialised by a subset component (when the subset component is added to a view). `download_attributes(AttributeList,SubsetComp)` allows a new base component to copy this subset component's data. The dependents of a base component include all of its subset components (related by `subset_rel`) and hence `update_dependents` is redefined for viewable base components.

IspelM defines base generalisation, client-supplier and classifier relationships as `base_gen`, `base_cs` and `base_cl` respectively. These are all defined as specialisations of `viewable_base_comp`.

6.4.4. Textual Base Components

Some base components have subset components which are rendered in textual display views. MVSL assumed MVisual supplied the text for a subset component with a textual display view component rendering. The MViews architecture, however, needs some mechanism for storing this text and for storing different textual renderings of the same base component. Base components which can have textual display view renderings are described by `text_base_comp`.

`text_base_comp` stores textual renderings of base components as *text forms*. Conceptually, a text form is a program graph stored as a single base component in the form of a

sequence of textual characters (i.e. a coarse-grained component storage). The data in this "graph" can be recovered by parsing the text and generating or updating other overlapping program graph information. Text forms and program graph information may be disjoint or overlap. For example, class definitions can be stored as text or as a program graph in IspelM, but method implementations are only stored as text forms and their structure recovered by parsing. Fig. 6.4. shows a text form and program graph from IspelM.
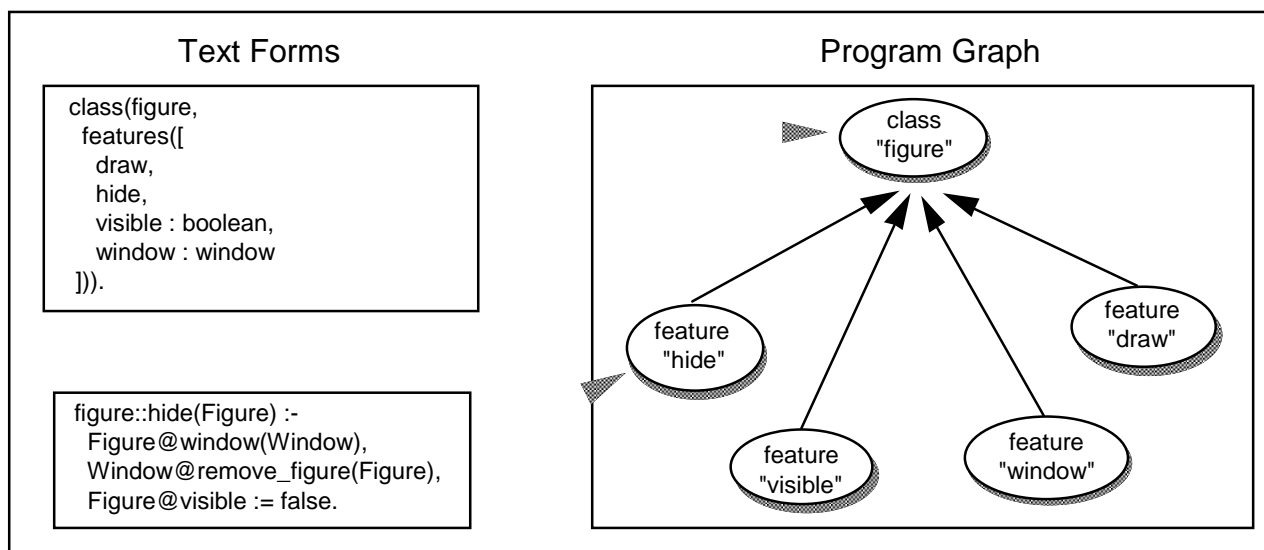
fig. 6.4.   Class and method text forms and a program graph from IspelM.

The MViews architecture places no constraint on using text forms or program graph representations for different tasks. The Dora data representation scheme (Wang et al 92) assumes a fine-grained PCTE storage scheme (i.e. all program components are stored as "program graphs"). MViews allows efficient, compact representation as text forms (which are edited using textual display views), program graph components (which are a finer-grained representation but usually less efficient in terms of memory and persistent storage), or a combination of both (possibly over-lapping).

`text_base_comp` provides methods to add, remove, find and update text forms (`add_text_form`, `remove_text_form`, `find_text_form` and `update_text_form`). In addition, `text_base_comp` provides methods for managing text forms when they are displayed in textual display views. These include creating new text forms in a view (create_text_view_form), displaying a text form in a view (display_text), and saving a text form's text to persistent storage (save_text). Textual display views can be associated with `text_base_comps` and methods for managing these views include storing a default textual view reference for the base component (`default_text_view`), displaying the default text view when selected (`text_selected`) and unparsing updates from the base component into the text form in a textual display view (`update_text`).

IspelM defines base classes and features as `base_class` and `base_feature`, both of which are specialised from `text_base_comp`.

6.5. Subset and Display Components

Fig. 6.5. shows the structure and methods defined for subset and display components. Subset components are implemented by `subset_comp` (i.e. are components of a subset view) while display components are implemented by `display_comp` (i.e. are components of a display view).
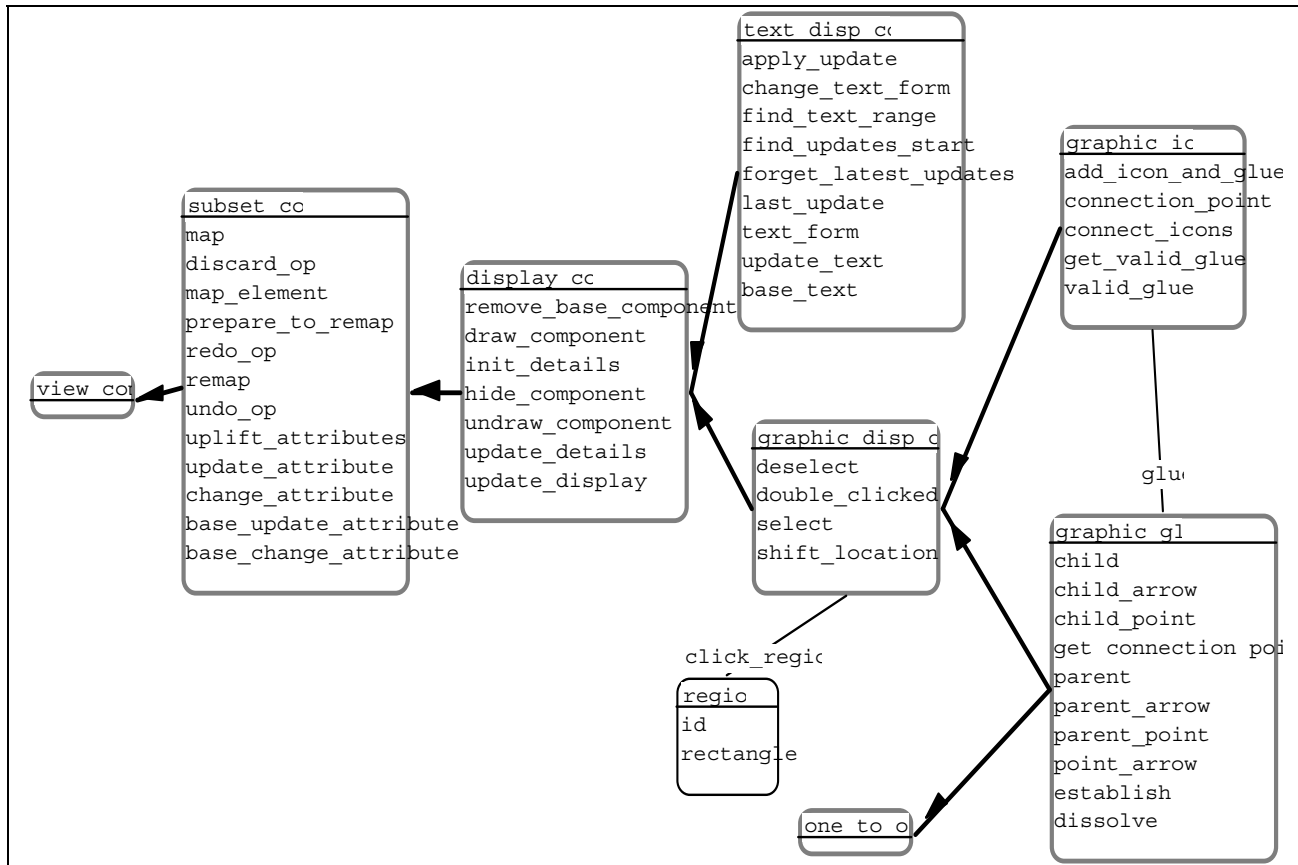
fig. 6.5.   Subset and display component structure and methods.

6.5.1. Subset Components

MVSL assumes a subset component will implement its own mapping operations to establish relationships to base components. As this is a common operation for all kinds of subset components, however, and since extra operations like remapping are required[16], `subset_comp` implements various mapping methods. These include mapping a subset

---

[16]Remapping is used when a programmer wants to change the details of a subset component and have it mapped to a different base component (i.e. **not** update the base component the subset component is already mapped to).

component to a base component (`map`), a deferred subset component-specific method that actually does the mapping (`map_component`), and remapping of subset components (`prepare_to_remap` and `remap`).

Subset components are usually created without being attached to a base component and then try to map themselves to an appropriate base component using `map`. Subset components may also not be mapped to base components if the base component they were mapped to has been deleted. Systems such as Dora (Ratcliffe et al 92), Unidraw (Vlissides 90) and the Object Design Editor (ODE) (Leidig and Mühlhäuser 91) require a view component to always be attached to a model component (and thus must automatically update view composition when models are deleted or no longer exist). This automatic update may or may not be what a programmer desires and may result in confusing or inappropriate view layouts and composition. MViews environments allow programmers to determine changes to subset views rather than automatically trying to update a view after an update.

Subset components implement a `record_update` method which sends their updates to their enclosing subset view. The subset component's view records the updates on itself and its components (much as a base component records its updates) and uses them to reverse or redo operations. Undo and redo of interactive manipulation is supported by sending a subset component or view update records it generated to undo or redo. Update records are discarded by subset views when they are no longer required for undo/redo and this process can delete subset components no longer required (i.e. that have been disconnected). This undo/redo mechanism is very generic (all basic operations are handled automatically) and extensible (new operations simply record update records or are built from a sequence of basic operations).

Subset components send update records to their display components by calling a subset component method `update_display(UpdateRecord)`. This method is implemented by display components, which are specialisations of subset components (see below). Display components and dialogues update subset components directly by sending them operations. Thus the MVSL subset component to MVisual display component update propagation mechanism is handled by `update_display`. The MVisual display component to MVSL subset component update propagation is handled by display components (and possibly dialogues) applying operations to subset components.

6.5.2. Display Components

Display components render a subset component in a graphical or textual form. As discussed in Section 6.2., the MViews architecture defines display components as specialisations of subset components. Thus a display component object actually includes all of the data and methods for the subset component it renders. This mechanism explicitly

defines the relationship between a subset component and its display component rendering (implicitly defined by MVSL and MVisual) as an inheritance relationship between `display_comp` and `subset_comp`.

A display component renders its subset component state using `draw_component` and removes this rendering using `undraw_component`. Display components can initialise and update their subset component state using `init_details` and `update_details` which typically use dialogues to modify subset component attribute values. `hide_component` deletes the subset component for a display component from its view while `remove_base_component` marks the base component for a display's subset component as removed.

A display component is drawn, undrawn or updated when its subset component sends it update records. The display component decides on an appropriate action to take by redefining the `update_display` method inherited from its subset component. A graphical display component can have one or more "sub-components" related to it which comprise part of its visual appearance. Sub-components are related to the display component using relationships and can thus be sent update records when the display component changes (they are dependent components). This allows a constraint system to be implemented to control related display components, similar to systems provided by LOGGIE (Backlund et al 90) and Unidraw (Vlissides 90).

6.5.3. Textual Display Components

Textual display components are a rendering of a `text_base_comp` text form. `text_disp_comp` records the id for the `text_base_comp` text form they render as `base_text`. `text_disp_comp` provides methods to change this base text form (`change_text_form`), expand updates into the text form's text from its base component (`update_text`) and apply updates expanded from the base component (`apply_update`).

Textual display components have a "range" in their view determined by an `updates_start` comment and the `updates_start` of their following component (or the end of the display view's window). Fig. 6.6. illustrates the text owned by text display component text forms.
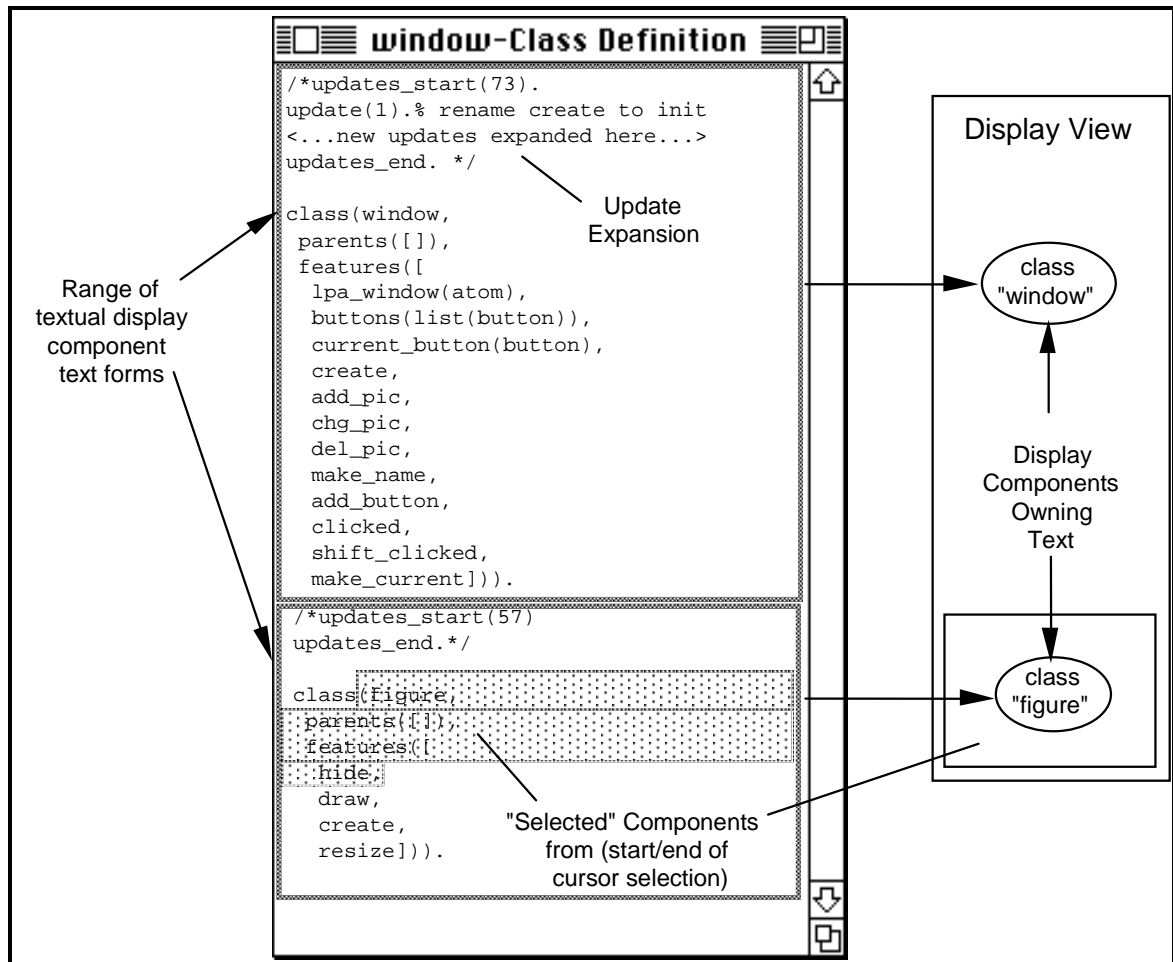
fig. 6.6. Associating window text with textual display components, inserting updates, and selecting textual display components.

The `updates_start` comment is also used to expand updates into a text view to inform programmers of changes to base information (possibly) not yet reflected in the text view. When a textual display component is updated, the update record is unparsed into a form readable by programmers and inserted after the last update in its `updates_start` comment. For example, the update record `update_attribute(Feature, feature_name, OldName, NewName)` might be unparsed into the form `% rename feature OldName to NewName`. The selected display components for a textual display view are those which have some of their text (including their `updates_start` comment and update records) within the cursor selection range of the text window. Fig. 6.6. illustrates how a text form is associated with display components, how updates are expanded into a text form after the `updates_start` comment, and how the selection range is determined.

Programmers usually treat textual program definitions as a series of tokens rather than structures at a low level of detail (e.g. expressions and to some degree control structures), suggesting a free-edited model of interaction rather than structure-edited as in the Mjølner (Minör 90) and Dora (Ratcliffe et al 92) environments. MViews assumes textual display components are "modified" by having their text changed by interactive text editing.

Textual display components can also be modified by selecting part of their text and applying menu-driven operations (see textual display views below).

Sometimes updates need to be "forgotten" and not expanded after parsing a textual display view, as a programmer is not interested in the change (as they are aware of it[17]). When MViews actually applies an update to a view to update the text for a component[18], MViews uses incremental token parsing and substitution to perform the update (see Chapter 7 for more details of this process).

IspelM defines class and method textual display components as `class_text` and `feature_text`, both specialisations of `text_disp_comp`. These render base program text forms for classes and features. Documentation text forms are provided for base classes and features and more than one documentation form per component is supported (by allowing multiple documentation text forms to be defined for a base component).

6.5.4. Graphical Display Components

MViews treats graphical display views as graph renderings made up of icons (nodes) and connector glue (edges). Both of these graphical display components can have sub-components representing part of a graphical display component. This representation scheme suffices for most MViews environments which treat diagrams as "boxes and glue". Further extensions could be made to provide similar capabilities to Unidraw, which supports arbitrary graphics, connectors and scalable glue (Vlissides 90).

`graphic_disp_comp` provides methods for selecting (`select`) and deselecting (`deselect`) graphical display component renderings and methods to interpret click (`double_clicked`) and drag (`shift_location`) editing operations on a display component. It also provides a list of "click regions" (as `click_regions`) which can be used to determine where inside the component rendering's border a click occured. The manner in which a graphical display component generates a rendering is assumed by the MViews architecture to be language and user interface toolkit-specific.

Graphical programming of object-oriented systems usually manipulates actual program structures (Ratcliffe et al 92) suggesting an interactive structure-oriented editing mode

---

[17]Usually this occurs when the change is made to this textual view and hence the programmer made the change and doesn't wish to be needlessly informed of it by an update record. This facility can be turned off, however, so programmers are always informed of any change thus ensuring no "unintentional" changes slip through.

[18]This "apply update" operation is typically done after programmer request but can be automatic.

rather than parsed as with GREEN (Golin and Reiss 90). MViews graphical display views provide tools which are used to interactively edit graphical display components and their subset component data.

6.5.5. Icons

Icons are connected by graphical glue which is attached at connection points on the icon border or inside its rendering. `graphic_icon` defines methods allow icons to be connected by glue (`connect_icons`), new icon and glue to be added (`add_icon_and_glue`), glue connection points to be determined (`connection_point`) and glue validity to be determined (`get_valid_glue` and `valid_glue`).

IspelM defines one graphical icon `class_icon` as a specialisation of `graphic_icon`. `class_icon` defines MVSL subset `class_icon` data and methods (such as `class_name` and `kind`) and provides methods which implement the MVisual interaction for class icons.

6.5.6. Glue

`graphic_glue` is used to connect icons and, in addition to being a `graphic_disp_comp`, is also a specialisation of the `one_to_one` relationship class (see below). Glue implements methods for determining icon connection points (`get_connection_points`, `parent_point` and `child_point`), arrow methods (`parent_arrow`, `child_arrow`, and `point_arrow`) and `establish` and `dissolve` methods for connecting class icons. Connector points on icons (such as Prograph dataflow entity pins (Cox et al 89) and Unidraw slots and pads (Vlissides 90)) can be implemented as sub-components, or icons being connected by glue can supply connection points for the glue using the `connection_point` method.

IspelM defines generalisation, client-supplier and classifier glue as `gen_glue`, `cs_glue` and `cl_glue`, all specialisations of `graphic_glue`.

6.6. Relationships

6.6.1. Relationships

MViews component relationships are modelled in the MViews architecture by the `relationship` class. This is specialised into relationships of different arities. Fig. 6.7. shows the relationship classes defined by MViews.
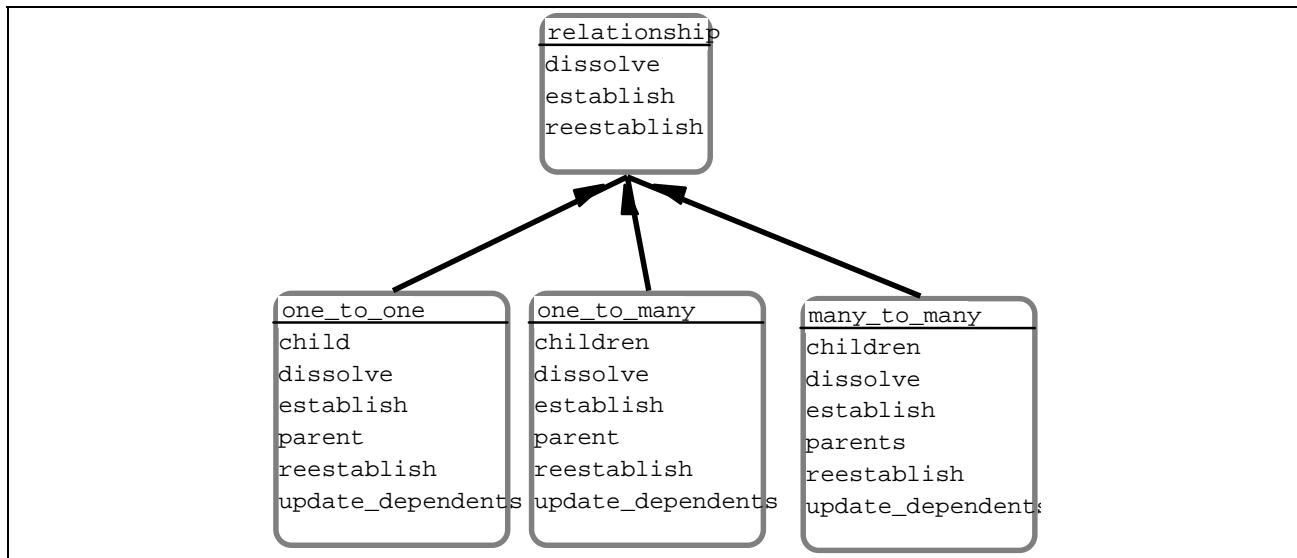
fig. 6.7.   MViews relationship class structure and methods.

`relationship` defines methods that equate to MVSL establish, reestablish and dissolve operations. Link relationships are modelled by the architecture as relationship components with no additional attributes, relationships or methods of their own.

6.6.2. One-to-one

One-to-one relationships relate a parent component to a child component. `one_to_one` implements `establish`, `dissolve` and `reestablish` for a single inter-component relationship. These methods produce update records of the form `establish(RelComp,Parent,Child)`. The dependents of a one-to-one relationship are its parent and child components plus any defined by `dependents` (inherited from `component`).

6.6.3. One-to-many

One-to-many relationships relate a parent component to one or more children components. `one_to_many` implements `establish`, `dissolve` and `reestablish` for single-parented multiple inter-component relationships. The dependents of a one-to-many relationship are its parent and children components plus any defined by `dependents`.

6.6.4. Many-to-many

Many-to-many relationships relate one or more parent components to one or more children components. `many_to_many` implements `establish`, `dissolve` and `reestablish` for a multi-parented multiple inter-component relationships. The dependents of a many-to-many relationship are its parents and children components plus any defined by `dependents`.

6.6.5. Subset/Base Relationships

MViews base components can have zero or more subset components. MVSL assumes these subset components provide update operations which translate base component updates to subset component updates and vice-versa. The MViews architecture, however, abstracts out this base component to subset component "update mapping" into subset/base relationships. A subset/base relationship implements both the base component to subset component update translation and the subset component to base component update translation. Subset/base relationships are many-to-many relationships thus allowing a subset component to be a composite "subset" of two or more base components.

Subset/base relationships act as an interface between a base component and its subset components and are dependents of both their base components and subset components. Subset/base relationships receive updates from their base components and update their subset components, if these subset components are interested in the base update, so the subset components are consistent with their base components. Fig. 6.8. shows some subset/base relationships for an MViews program and its views.
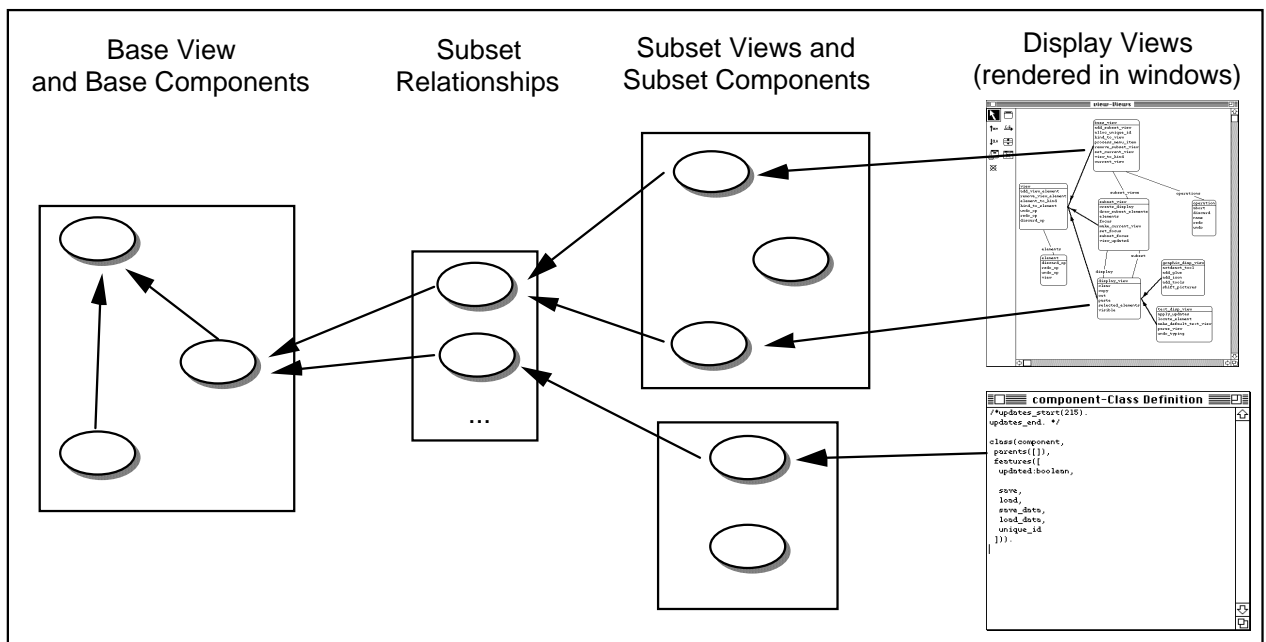
fig. 6.8.   Some subset/base relationships connecting base components and subset components.

All subset components of the same kind for a base component are linked to a single subset/base relationship object for the base component. For example, three class icons for a base class in IspelM are linked to the same class icon subset/base relationship. This allows for efficient update processing for each kind of subset component as only one subset/base relationship processes a base update record for all subset components of the same kind. Lazy propagation of base updates to subset components can also be supported by recording base updates against subset components for later processing. This is useful

for subset views that are hidden and hence there is no point in immediately updating their subset components and thus having the display components re-rendered (usually an expensive operation as graphical user interfaces consume much processing power (Dannenburg 91, Vlissides 90, Backlund et al 90, Minör 90)).

Subsets relationships are created by subset components and maintain relationships between all their base components and subset components. Fig. 6.9. shows the subset/base relationship structure and methods.
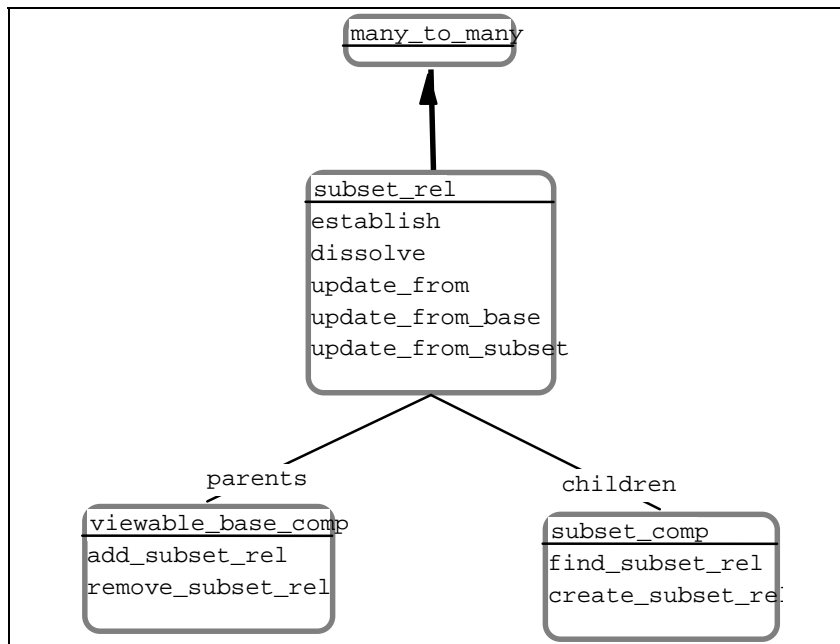
fig. 6.9.   Subset/base relationship structure and methods.

The parent components of a `subset_rel` are `viewable_base_comp` objects (denoted by the `parents` aggregation relationship in fig. 6.9.) and the child components are `subset_comp` objects (denoted by the `children` aggregation relationship in fig. 6.9.). `subset_rel` redefines `establish` and `dissolve` which create and remove relationships between base components and subset components. `subset_rel` receives update records from its base components and subset components. It calls `update_from_base` and `update_from_subset` appropriately to map an update into a corresponding change in the related components.

The separation of subset components and base components by subset/base relationships allows the same subset component to be connected to different (or the same) base components using different subset/base relationships which transform base and subset update records differently. For example, a bar graph display component illustrating the run-time performance for part of a program could model any kind of base collection component (hashtable, list, graph) using a different subset/base relationship to interpret the base component data and update records (see Chapter 9 for further details). This bar graph display component could also be used to animate a sorting algorithm using a different subset/base relationship which transforms sorting algorithm update records into

bar graph operations (see Chapter 9). A subset/base relationship performs a similar task to "watcher" objects in Tarraingim (Noble and Groves 93).

IspelM defines subset/base relationships for classes, features, generalisations, classifiers and client-supplier relationships. Features are modelled as a special case of client-supplier glue (a named and typed aggregate connection at the code (implementation) level) and a client-supplier subset/base relationship translates between base features and base client-suppliers to client-supplier glue.

6.7. Views

6.7.1. Views

MViews uses views to group base program graph components (base views), group subset graph components of these base components (subset views) and to group renderings of subset components as display components (display views). Fig. 6.10. shows the structure and methods supported by different specialisations of views.
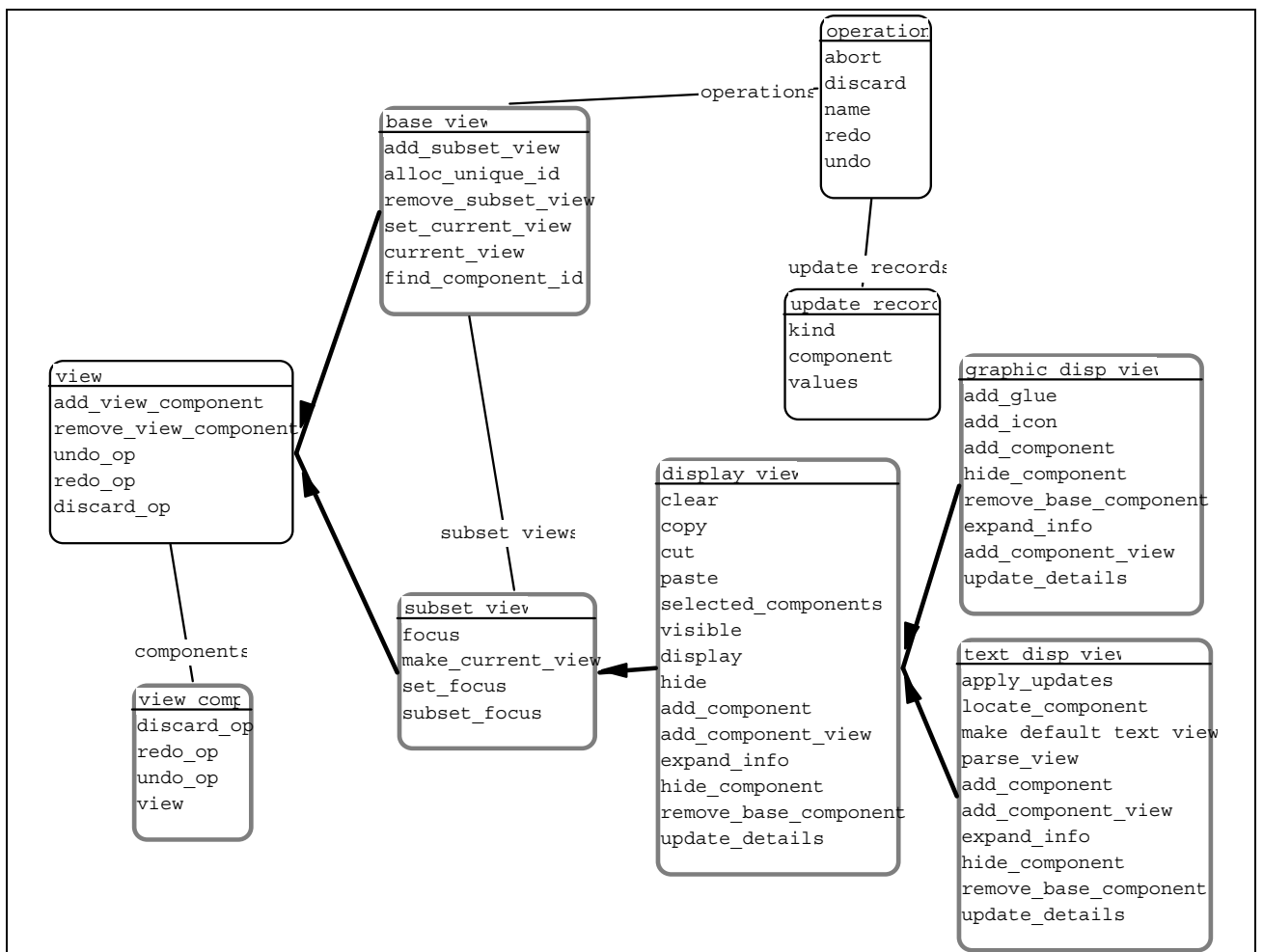
fig. 6.10. View structure and methods for MViews.

Views maintain a one-to-many relationship to the components they enclose (`components`) and provide methods to add and remove these components (`add_component` and `remove_component`). Both views and their components support undoing, redoing and discarding of update records by `undo_op(UpdateRecord)`, `redo_op(UpdateRecord)` and `discard_op(DoneOrUndone, UpdateRecord)`.

6.7.2. Base Views

A base_view corresponds to an MVSL base view declaration. `base_view` maintains a one-to-many relationship to all subset view objects held in memory with `subset_views`. A *current view* (typically the subset view whose display view window is the front window) is maintained in `current_view` and changed with `set_current_view(SubsetView)`. Base views allocate unique component id values for views and their components using `alloc_unique_id` (so all component ids are unique for a given base view). A base view also supports look up of component objects using this id with `find_component_id`.

`base_view` maintains a list of "history" operations used to implement undo/redo in `operations`. An `operation` stores a list of update records generated by components for each interactive editing operation (which may generate several update records) performed on display views. Subset views and their components record the update records they generate by sending them to their base view (using `record_update`). The base view can then undo or redo these interactive operations on programmer request by sending the update records generated by the operation to the creating component's `undo_op` or `redo_op` method as appropriate. `discard_op` is used when operations are no longer required (typically a limited number are kept by the base view, as with Unidraw commands (Vlissides 90)).

IspelM defines one base view to store object-oriented program data (which IspelM calls a `program`). This class extends `base_view` to support component kind-specific look-up tables to locate classes, features and predicates by name.

6.7.3. Subset Views

A `subset_view` contains several `subset_comps` which comprise a subset program graph, and hence `subset_view` corresponds to an MVSL subset view declaration.

A subset view has a focus (the base component that "owns" the view), referred to by `focus`, and a subset component that is linked to this focus component, referred to by `subset_focus`. The `subset_focus` component can not be removed from the subset view unless another subset component is designated the focus for the view (by set_focus). A subset view can be made the current view by calling `make_current_view` which indicates that editing operations are to be applied to this subset view's display view.

6.7.4. Display Views

A `display_view` renders a `subset_view` in a textual or graphical form and hence is equivalent to the MVisual notion of a display view. Display view components render subset components and the display view is assumed to group these renderings in a window (assumed to be provided by the implementation language/user interface toolkit for MViews).

`display_view` is a specialisation of `subset_view` and hence behaves as a subset view component as well as a display view component. A display_view may be shown or hidden (i.e. its window shown or hidden) by `display` and `hide` and this display status is indicated by `visible`.

A `display_view` provides methods for manipulating the display components it encloses. These methods include `cut`, `copy` and `paste` of selected display components, adding display components (`add_component`), hiding the subset components for selected display components (`hide_component`) and removing these subset component's base components (`remove_base_component`), expanding selected display component information from the base view (`expand_info`), updating a component's details (`update_details`), and adding a new view for a display component (`add_component_view`).

6.7.5. Textual Display Views

`text_disp_view` is a specialisation of `display_view` and is used to render textual display components for a subset view's components. A `text_disp_view` thus corresponds to an MVisual textual display view. Textual display views are parsed to update the base component information of their subset components using `parse_view`. A textual view is composed of a linear sequence of text forms distinguished by `updates_start` comments. A textual display component can be located given a cursor position in this text by `locate_component`.

Textual display component text forms are manipulated by modifying their text using free-editing operations. Menu-driven commands are used to modify the display components and their subset components (such as `add_component`, `hide_component`, etc.). When a textual display view's subset view becomes the current view the textual display view unparses any updates on its display components (i.e. inserts a human-readable form of new base component update records into the text form text of its display components). These updates can be applied to the text forms by `apply_update`.

IspelM defines one textual display view `class_text_view` which is a specialisation of `text_disp_view`. This view is used to render class and method code and for viewing base component documentation.

6.7.6. Graphical Display Views

`graphic_disp_view` is a specialisation of `display_view` and is used to render graphical display components for a subset view's components. A `graphic_disp_view` thus corresponds to an MVisual graphical display view.

Graphical display views maintain a one-to-many relationship to graphical display components (either icons or glue). New icons and glue are added to the display view with `add_icon` or `add_glue`. Graphical display view components are manipulated via direct manipulation using tools (which call `hide_component`, `expand_info`, etc.) or by dialogues.

IspelM defines one graphical display view `class_diagram_view` which is a specialisation of `graphic_disp_view`. A class diagram view is used to represent and manipulate class diagrams made up of class icons and generalisation, client-supplier and classifier glue.

6.7.7. View Composition and Layout

The composition of subset views is controlled by programmers adding and deleting components to and from the subset view (via display view manipulations). Automatic expansion of data from base components can be supported by subset components interpreting base component update records or expanding base component information when requested by programmers. For example, an IspelM class icon could expand all the feature names of its base class by adding new feature names when its base class is updated.

The layout of display views is controlled by programmers rather than automatic graph layout algorithms, although these could be implemented by display views. ODE supports automatic graph layout and expansion into views (Leidig and Mühlhäuser 91), as does Graspin (Mannucci et al 89). EDGE (Newbury 88) also supports automatic layout with users being able to modify graphs to suit their requirements using constraints.

MViews allows programmers to determine both the layout and composition of views and informs them of changes to view data (by expanding update records or changing icon and glue graphical appearance). It does not attempt to modify or correct subset view inconsistencies nor graphical view layouts. MViews allows programmers to make appropriate modifications in response to indications of the updates affecting subset view components.

Automatic layout often produces unsuitable layouts when applied to frequently changing graph-based diagrams (Paulisch and Tichy 90). MViews subset and display views are typically used for applications which have a long life-span (for example class diagrams and code) and automatic view layout for these applications would generally produce large-scale re-layout which may confuse or hinder programmers (Paulisch and Tichy 90).

For this reason graphical display views do not currently support an automatic layout algorithm. There is no reason why one should not be implemented for them, however, as most algorithms use graph topology and icon and glue sizes, both of which can be determined from MViews graphical display components and their renderings. Such layout algorithms would be of great assistance for display views which are generated frequently, such as for call graphs (Reiss 90b) and object debugging traces (Kleyn and Gingrich 88).

6.7.8. Application Framework

Each MViews system may have one or more separate base views under construction. Subset and display view components from one program may be copied to another using display view cut and paste operations. Base component copying must be implemented in an application-specific manner, as base components can have very complex inter-component relationships, whereas inter-component relationships for subset and display components are generally restricted to their enclosing view components and base components. Each running MViews system has one `application` component which keeps track of all base views. The application creates new programs or reloads old programs from persistent storage.

6.8. Operations and Update Records

6.8.1. Operations and Update Operations

As discussed in Section 6.2., MVSL operations and MVisual updates are implemented by class methods in the MViews architecture. Basic operations are implemented by methods defined by the MViews architecture classes. For example, `component` defines `update_attribute( AttributeName, NewValue)` to implement the update attribute (`Comp.AttributeName:=NewValue`) operation from MVSL. `display_view` defines the method `display` to implement the MVisual display update on "view" graphical entities.

Component-specific operations are defined by implementing new methods for the specialisations of MViews architecture classes which use the basic operation methods defined for MViews. For example, an `add_feature(NewName,NewKind,NewType)` operation for base classes in MVSL could be implemented by the method `add_feature(NewName,NewKind,NewType)` for `base_class` (where `base_class` is specialised from `base_text_comp`). This `add_feature` method would create a new `base_feature` object, establish a `class.features` relationship to the feature's owning class and initialise the new feature by calling an `init(NewName, NewKind, NewType)` method for `base_feature` (which uses `update_attribute` to initialise `feature_name`, `kind` and `type_name` for `base_feature`).

The MViews architecture assumes an `update_from` method is implemented by components to determine a component's response to an update record. This `update_from` method equates to update operations in MVSL and, given an update record, will apply operations to a component which defines its response to an update record. This method might be implemented by a case-statement on an update record's kind and values (to determine a sequence of methods to call to implement the update operation). The Snart implementation of the MViews architecture uses a Prolog-style pattern matching on update records, which are represented as terms (see Chapter 7).

6.8.2. Update Record Generation and Storage

After applying an operation to MViews components, the method implementing the operation will generate an update record and call `record_update(UpdateRecord,UpdateName)` to propagate (and possibly store) the change this operation has caused to the component.

Fig. 6.11. shows uses the `update_attribute` operation to illustrate how an operation generates an update record, how this update record is propagated to dependents of the generating component by `record_update`, and how it is stored against the component using `store_update`. This example uses base features and classes from IspelM to also show how updates can be passed from a generating component to another component for storage and propagation.

1. The `update_attribute` method is called for a base feature. `update_attribute` changes `feature_name` to "calcValue".

2. `update_attribute` generates an update record to describe the change done to the base feature and calls `record_update` with this update record to indicate a change has taken place.

3. `record_update` broadcasts this update record to dependents of the base feature (in this case its owning class and feature subset/base relationships).

4. Upon receiving an update record from one of its sub-components (in this case a base feature), `update_from` for a base class uses `store_update` to store the update record received to document the change its sub-component (and hence itself) has undergone.

5. `update_from` for the base class also propagates the sub-component update record to its dependents (in this case a class subset/base relationship).

6. The class subset/base relationship receives the update record from its base component and sends it to its subset (in this case a class display component for a text view).

7. The class text display component unparses the update from its base component into its text form's text. For some updates, such as renaming of the class, this may also generate operations (to change the display component's `class_name` to that of its updated base class name). This may generate further update propagation in a similar manner to step 1.
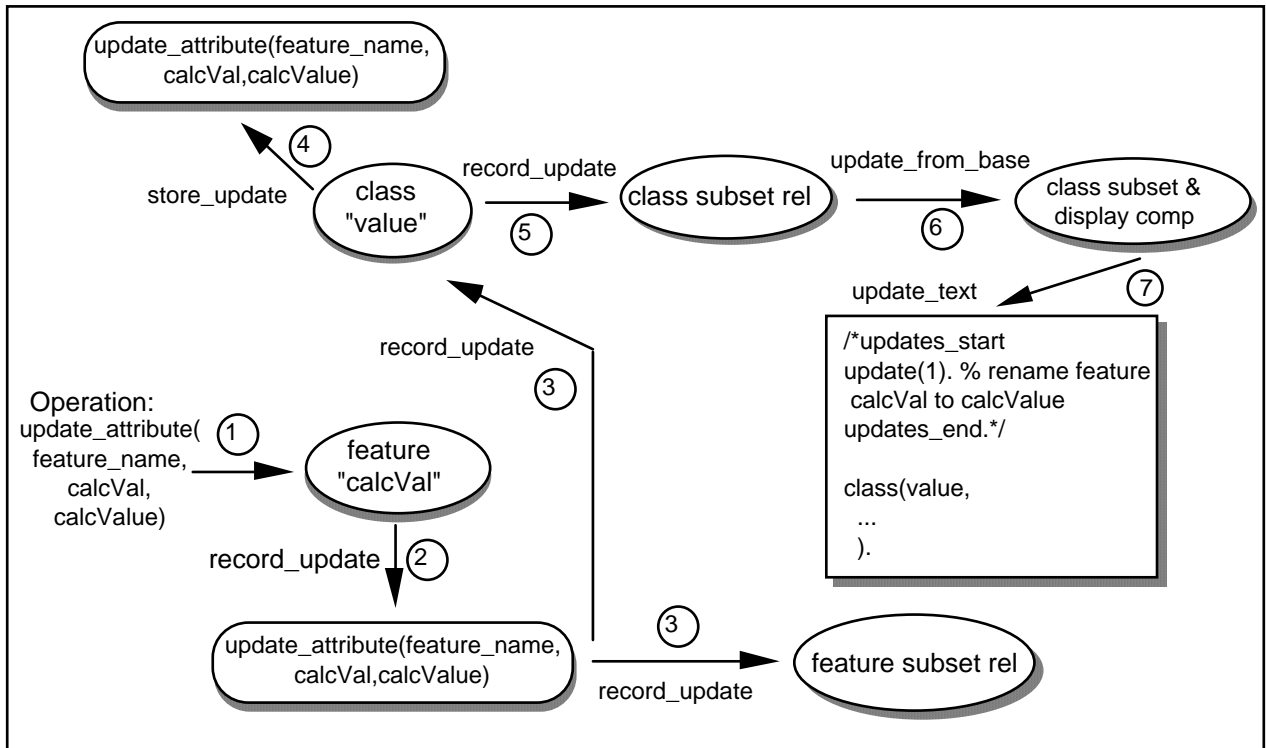
fig. 6.11. Generation, propagation and storage of update records by `update_attribute`.

6.8.3. Composite Update Records

Update records are generated by basic operations in MVSL and the corresponding MViews architecture methods generate similar update records. For example, `CompID.delete` generates update records of the form `delete_component(CompID)`, `CompID.establish_rel(Kind, CompID, Child)` generates `establish(NewRel, CompID, Child)` and so on.

Methods defined for specialisations of MViews architecture classes can also generate their own "composite" update records, if required. For example, the method call `Icon@shift_location(NewX, NewY)` for a `graphic_icon` object generates a `shift_location(CompID, DX, DY)` update record when an icon is dragged interactively. This corresponds to generating two `update_attribute` records `update_attribute(CompID, X, OldX, NewX)` and `update_attribute(CompID, Y, OldY, NewY)`. These two basic update records are still produced for any dependents who only want to be informed of the change in state of one of an icon's location attributes. The composite update, however, is generally more useful for sub-icons and glue which only want to know the change in an icon's location and hence need only provide update response processing for a `shift_location`

update record. Chapter 10 discusses extending this composite update record system to provide automatic composite update record generation from basic update records.

6.8.4. Update Record Propagation

Update records generated by MViews methods are propagated to all dependents of the generating component. These dependent components may invoke further methods in reponse to these update records which in turn may generate further update records for propagation. Fig. 6.12. illustrates how an operation method sent to a display component is propagated to change the MViews program state. The steps in the propagation are:

1. a. Textual view is updated by typing, and parsing gives parse tree to subset/base relationship (located via display/subset component associated with text form) and subset/base relationship updates base component.

   b. Graphical or textual view component is updated by direct manipulation, menu or dialogue.

2. Method call for display component translated into method(s) call for subset component (if necessary) by display component.

3. Method call for subset component generates update records which are sent to its dependents (subset/base relationships) which interpret these update records with `update_from` (which calls `update_from_subset`).

4. Subset/base relationship's `update_from_subset` method translates subset component update records into base component method calls if the base components are affected by these subset component updates.

5. Base component stores the update record (if required) using `store_update`.

6. Dependent components of base component sent update records generated by base component's methods (these dependent components include subset/base relationships of the base component, if any).

7. Subset/base relationships interpret base update records and determine if their subset components need updating. `update_from_base` calls subset component methods if necessary.

8. Subset component's methods record their update records and `record_update` sends these update records to display components by calling `update_display` for the display components.

9. Display components either re-render themselves in response to the update records sent to `update_display` or expand these update records into a human-readable form to indicate changes affecting them (using `update_text`).
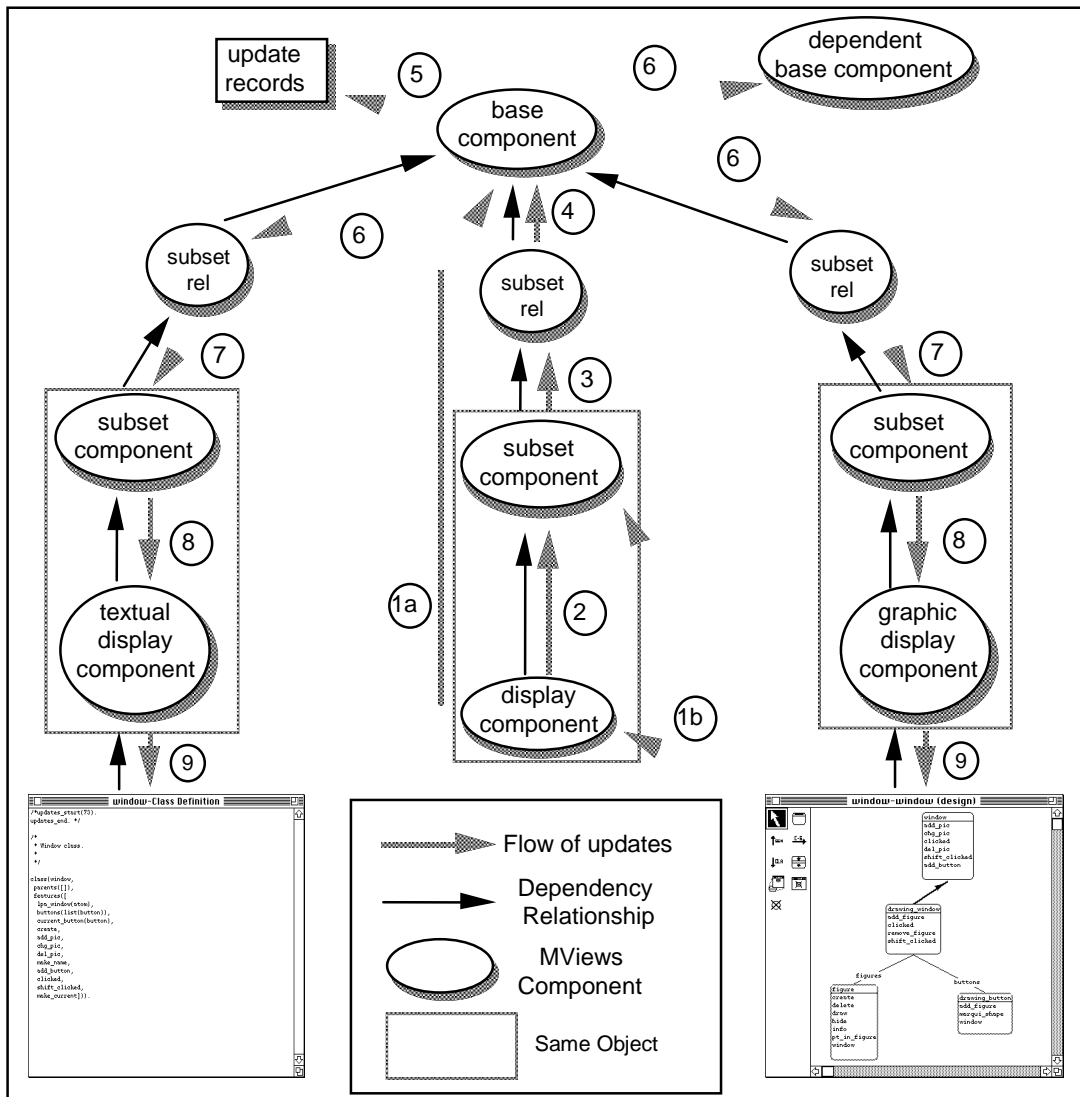
fig. 6.12. Flow of change after a display operation is performed.

After applying an operation to a subset (or display) component, the component's enclosing view records the update record generated to implement undo/redo. This undo/redo mechanism is currently sequential and global i.e. updates must be undone in reverse order to how they were applied and must be undone across all views (thus MViews supports undo as a history of operations, similar to Unidraw). A useful extension would be to allow updates to be undone and redone in an arbitrary order, and "generic" updates to be applied as a group to a view, as in PECAN (see Chapters 7 and 10 for more discussion of such facilities).

6.8.5. Constraints, Semantic Attribute Recalculation and Lazy Updates

When a component is modified by a method call (operation), constraints can be checked to ensure the operation is valid. These can be defined by over-riding the operation method in a sub-class or by over-riding `record_update` in a sub-class. The MViews architecture supports a facility to abort partially applied operations if constraint checking fails. Any update records stored by an operation object are then deleted and their effects reversed (in the same way as undo). For example, if an IspelM base class is renamed to the same name as some other class, an error message can be displayed and the rename (update attribute) operation aborted.

As all dependent components are notified of a component update, any dependent part of their state can be recomputed to reflect the change (attributes recomputed, constraints checked or the component marked for deletion). This provides a data-driven/event-driven mechanism similar to attribute grammars (Reps and Teitelbaum 87) and Garlan's dynamic view updates (Garlan 87) and Wilk's lazy consistency management (Wilk 91). Update records sent to a component could also be stored for lazy interpretation when a value affected by the change is required, or this update record interpretation could be deferred until a programmer requires computation to be performed (i.e. demand-driven evaluation).

A component receiving an attribute update record can use `update_from` to update its own attributes which depend on the updated attribute (in the other component or even itself). This provides a data-driven scheme where object dependency subsumes attribute-dependency in a similar manner to attribute grammars. An extension to this approach could provide a table which maintains a list of `<Attribute,<Object,Attribute> list>` pairs for all of attributes which depend on other `<Object,Attribute>` values. This would support data-driven attribute recalculation in a similar manner to (Reps and Teitelbaum 87), possibly with an incremental recalculation algorithm (Hudson 90).

The MViews architecture allows update records received by a component to be stored. The component can then be marked as "has updates needing actioning". Combined with the above approach of attribute dependency, this allows lazy and demand-driven attribute updating and object dependency propagation schemes to be implemented. Accessing an object attribute would cause any unactioned updates to be processed, possibly computing a new value for the required attribute. Alternatively, updates can be processed when received, but mark dependent attributes (and, transitively, any of these attribute's dependents) as "need recomputation". Accessing a "needs recomputation" attribute would cause it to be recomputed before being returned (i.e. it is lazily re-evaluated).

Cyclic dependencies are permitted but no automatic detection of this is currently assumed. Unidraw uses a similar model for a dataflow state variable propagation

mechanism (Vlissides 90) but checks for cycles by storing a list of all visited components resulting from an initial attribute update. Such a model could be employed by MViews when an attribute is changed to detect cyclic dependencies and either stop the change propagation or flag an error.

6.9. Discussion and Future Research

In this section we evaluate the MViews architecture with respect to program representation and manipulation, view and view component representation and manipulation, and operation and update record support. Possible future extensions to this architecture are also discussed.

6.9.1. Components

This `component` class captures the basic notion of an MVSL component. This representation works well for defining basic operations, component attributes and for representing update records. It also provides a flexible scheme for composing component-specific operations which make use of basic operations (and other component-specific operations) via method calls. This scheme does loose some of the abstractness (though not expressiveness) of MVSL when describing relationships and update operations. Relationships must be represented as a combination of attribute value (with a relationship component type) and specialisations of (or parameterised) relationship classes.

For example, `class.features` must be represented as an attribute of type `one_to_many` relationship component. The value of `class.features` is thus a one-to-many relationship to `feature` and `class.features.children` gives the list of features owned by a class. Establishing and dissolving such a relationship requires `establish_rel` and `dissolve_rel` for a component to be over-ridden to call `establish` and `dissolve` for `one_to_many` for `class.features`. The MVSL description of `class.features` from Appendix D allows such a relationship to be expressed and manipulated more succinctly.

Similarly, to implement update operations, the MViews architecture must over-ride `update_from` and implement a case-based selection on update record kinds and values. This is less abstract than describing update record responses with MVSL but once again does not actually lose any power of expression (all update responses from MVSL can still be described using the MViews architecture).

6.9.2. Base Components

The MViews architecture describes MVSL base elements and relationships by specialisations of `base_comp`. These specialisations define extra attributes and methods for supporting text form management, subset/base relationship, component and view management, and delayed base component removal. MVSL components are thus not only

described as specialisations of `base_comp` but also as specialisations of base components with extra characteristics as appropriate.

One disadvantage of this approach is that an MVSL base component must be described by an MViews architecture class specialisation which "knows" about its viewing mechanism. For example, MVSL does not require a base class to know whether it has text forms or any subset components at all but assumes MVSL subset views and MVisual display views and components define these notions. A further disadvantage is that the MViews architecture currently assumes components which have text forms (which may or may not have textual view renderings) can also have graphical view forms. A better structure for the base component classes to solve these problems might be to specialise `base_comp` to `graphic_view_base_comp`, `text_form_base_comp`, and `text_view_base_comp`. MVSL base components could then be modelled as specialisations of one or more of these classes (i.e. multiple inheritance) as appropriate or even all three (if no assumptions about the kind of base component viewing are made).

6.9.3. Subset and Display Components

The MViews architecture describes MVSL subset components as specialisations of `subset_comp` and MVisual display components as specialisations of `display_comp` (which are themselves specialisations of `subset_comp`). This scheme works well for most applications and provides a concrete, implementable relationship between a subset component and its display component using specialisation. For environments where a subset component can have several different display components this scheme is not as abstract as MVSL and MVisual. A specialisation of `subset_comp` can be defined with multiple specialisations from itself and `display_comp` to represent this situation. This requires more effort than an equivalent MVSL/MVisual specification as methods may be over-ridden by both the `subset_comp` specialisation and `display_comp` specialisations (and thus require some form of reconciliation under multiple inheritance).

6.9.4. Views

MVSL views are defined by the MViews architecture as specialisations of `view` with `display_view` defined as a specialisation of `subset_view`. As with subset and display components, this subset and display view representation scheme works well for modelling most environments described with MVSL and MVisual. Specialising `display_view` into `text_disp_view` and `graphic_disp_view` provides a natural way of expressing the kind of rendering the view supports. The architecture must provide appropriate methods for display view interaction, however, which define MVisual component manipulation assumptions (for example, display component addition and details updating).

6.9.5. Relationships

`relationship` models MVSL relationships and is specialised to relationships of one-to-one, one-to-many and many-to-many arities. Subset/base relationships abstract out the base to subset and subset to base translation of update records into component operations (method calls). Partial automation of this update propagation can be provided by a mapping of base component attributes and relationships to subset component attributes and relationships and vice versa.

Extending this relationship representation scheme to explicitly define part-of relationships would assist in implementing automatic update record propagation via transient dependencies (Wilk 91). The MViews architecture assumes an object-oriented implementation language will support multiple inheritance (so the `relationship` classes can be reused in conjunction with other `component` specialisations). This could be modelled by a relationship component attribute in such specialisations (i.e. relationship objects whose methods are called by the component specialisations they are created for) but this would not be as natural as the architecture's approach.

6.9.6. Operations and Update Records

MVSL basic, component-specific and update operations are modelled as record_update and update_from methods. This update record propagation mechanism is very flexible and provides an efficient method of implementing view updating and attribute dependencies. Storage of update records allows components to document their changes and supports a generic undo/redo mechanism.

6.10. Summary

The MViews architecture has been developed to abstract out the common features of environments which support multiple textual and graphical views of a program with consistency management. This architecture provides a set of reusable components based on the concepts of Chapter 5 which allows MVSL and MVisual environment specifications to be modelled as specialisations of appropriate MViews architecture classes. IspelM can be defined in terms of this MViews architecture by specialising classes to describe program representation as graphs (base components and view), views of these program graphs (subset and display components and views), and operations and update responses for each kind of component (using methods and method over-riding). The MViews architecture provides additional classes and class attributes and methods for supporting concepts of MViews environments. These include base component classes with methods to manage views and text forms, display components for textual display components (including update unparsing and application) and graphic display components (including

icon and glue support), and display views (including textual and graphical renderings of program fragments).

Novel aspects of the MViews architecture include its use of the component class to model generalised object dependency graphs. These graphs are used for representing program structural and semantic information in the same manner. Subset graphs are represented in the same way as base program graphs and display views and components are defined as specialisations of subset views and components. MViews solves the textual view consistency problem in a novel manner by unparsing update records stored against base components and can automatically apply some of these updates to text forms on programmer request. Components can determine their response to update records sent via the object dependency mechanism and this can be used for general object dependency (attribute recalculation etc.), constraint maintenance, and efficient subset and display view updating. Storage of update records supports a generic undo/redo facility and documentation of program component changes.

This object-oriented architecture for MViews can be used to create an object-oriented design for environments specified with MVSL and MVisual. Such a design requires an implementation to produce an environment and thus an implementation for the MViews architecture is necessary. Chapter 7 discusses an implementation of the MViews architecture in Snart which produces an object-oriented framework for MViews. Chapter 8 uses this architecture to produce a design for IspelM and uses the Snart framework to produce an implementation of this design for IspelM and SPE.

Chapter 7

An Object-Oriented Implementation of MViews

Chapter 6 described an object-oriented architecture for MViews systems based on the model and specification languages in Chapter 5. To construct executable environments this architecture must be implemented using a programming language. In this chapter an implementation of MViews as an object-oriented framework in Snart is described. This framework provides a set of classes that support the component structures and operation methods described in Chapter 6. New environments specialise these Snart classes to implement their own program representation, subset and display views and components, user interface, persistency management, and interfaces to existing compilers and run-time systems and/or the static and dynamic semantics of a language.

The reasons for implementing MViews as a Snart framework and advantages of choosing this implementation language over comparable approaches are discussed. The implementation of each type of component from Chapter 6 is described with particular attention to Snart-specific implementation decisions. The framework is evaluated with future extensions and alternative implementation approaches for MViews briefly discussed. Chapter 8 reuses the object-oriented architecture of Chapter 6 to model IspelM and reuses the Snart framework to implement this IspelM model and to specialise IspelM to produce SPE.

7.1. A Snart 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 (defined in Chapter 5). From this specification MViews class hierarchies were derived for the object-oriented architecture of Chapter 6. Class responsibilities and services were determined from MVSL and MVisual operations and updates and the MViews and IspelM architectures implemented.

These design and implementation processes were concurrent with feedback between each. This evolutionary software development implied a need for a language supporting

experimental programming. Some aspects of both MViews and IspelM were not easy to determine without a prototype implementation (particularly user interaction through display views and persistency management for programs). The architecture used to model MViews in Chapter 6 is object-oriented, assuming multiple inheritance, encapsulation of data and behaviour, and polymorphism. It is thus more natural and easier to implement this architecture in a language supporting such concepts, rather than non-object-oriented languages, such as C or Pascal.

Prototype implementations of MViews and IspelM were initially attempted using Quintus Prolog's ProTALK on a DECstation 2100 (see Chapter 3 and (Quintus 91)) and THINK C (a C++-like language on the Macintosh (Symantec 91)). The object-oriented facilities of ProTALK were not at all satisfactory and difficult to use and its environment very rudimentary. THINK C provided only simplistic object extensions to C and was difficult for prototyping due to its strongly typed nature (forcing many long compilations when class hierarchies were changed). Both languages provided little high-level support for prototyping or constructing user interfaces. Quintus Prolog's X-windows interface facilities were very low-level while THINK C provided the THINK Class Library (TCL) framework for accessing the Macintosh Toolbox facilities. Both required major effort to build even simple user interfaces compared with LPA MacProlog and these interfaces proved much less flexible or extensible.

As discussed in Chapter 3, Snart was designed to be a simple language combining Prolog's untyped, logic programming facilities inside an imperative object-oriented structure. Snart was designed to be fast in execution time, have efficient object storage, and have an extended environment for object-oriented programming. Its simplicity compared with other available prototyping languages (Smalltalk (Goldberg and Robson 84), CLOS (Keene 89) and ProTALK) is an advantage together with complete control over Snart's implementation. For some aspects of our work Snart itself evolved to support a dynamic object tracing facility for dynamic program visualisation and visual debugging (see Chapter 9), and object persistency management to experiment with transparent MViews program persistency.

Although Snart is essentially untyped, Snart program structures could be ported to strongly-typed languages such as Eiffel, C++ or Kea. The high-level support for building graphical user interfaces in LPA MacProlog and incremental compilation inside the LPA environment help make Snart an excellent rapid-prototyping language. While MViews could be implemented in other object-oriented languages, such as Protalk, C++ (Stroustrup 86), Smalltalk or CLOS, we determined that Snart would be a suitable implementation language for an experimental prototype.

7.2. MViews Framework Complexity

Table 7.1. shows the breakdown of code in MViews. In addition to implementing the MViews architecture classes from Chapter 6 the Snart implementation of MViews provides groups of Prolog predicates which implement:

- basic persistency management (for reading and writing terms and objects to files)
- LPA MacProlog menus and dialogues and support predicates
- unparsing predicates and classes for data structure support
- class interfaces to LPA MacProlog window processing predicates

MViews Components	Lines
component	370
view_comp	280
relationships	231
subset_rel	212
base_comp	120
viewable_base_comp	328
text_base_comp	338
subset_comp	397
display_comp	101
text_disp_comp	390
graphic_disp_comp	169
graphic_icon	196
graphic_glue	381
view	172
base_view	388
subset_view	570
display_view	267
text_disp_view	532
graphic_disp_view	985
application	275
dialogues and menus	581
persistency support	456
undo/redo support	550
misc. (unparsing, data structures)	454
Total:	8743

table 7.1. Complexity of the MViews implementation.

7.3. Components

7.3.1. Component State

All component classes from the MViews architecture are implemented as Snart classes. A Snart class interface defines generalisation classes, attributes, and methods (operations) for each kind of component. Snart allows attribute and method names to be Prolog variables which are bound at run-time. For example, a call of the form `Comp@Attribute:=Value` is valid if `Comp` and `Attribute` are bound to appropriate values at run-time. The Snart framework implements component attributes as object attributes, rather than a one-to-many relationship to `attribute` objects, as defined by the architecture in Chapter 6. This attribute representation is both space and execution time efficient as a component object needs no extra objects to represent its attribute values. Operation methods are implemented as Snart methods with method arguments supplying information used by the operation. Relationship component objects are accessed via object attributes with `establish_rel` and `dissolve_rel` methods supplied to manipulate component relationships. One-to-one relationships can also be implemented by Snart object attributes directly referencing other component objects.

The Snart framework currently assumes component objects are referenced by their Snart object id or an application-specific unique component identifier. Component classes can define a `unique_id` method which is used by the `base_view` to look-up components for a program. When a component is reloaded from persistent storage a new Snart object is created for it. `unique_id` provides a component reference which exists across different reloads of a component as different Snart objects, and thus can be used by relationships and attributes representing relationships to refer to components.

7.3.2. Update Records

Update records are represented as Prolog terms of the form `UpdateKind(Component, Value`$_1$`, ..., Value`$_n$`)`. Terms are used rather than objects for efficiency and because all update record processing is provided by components rather than the update record itself. Thus update records need not be represented with an object-oriented structure (i.e. we need only store data and not data and behaviour).

Fig. 7.1. shows some Snart code from the MViews `component` class implementation. The `component` class name is prefixed by "`mv_`" to distinguish it from classes belonging to specialisations of MViews.

```
abstract_class(mv_component,
   parents([]),
   features([
      dependents:list(mv_component),
      updated:boolean,
      id:integer,
      ...
      get_attribute,
      update_attribute,
      ...
      record_update,
      update_dependents,
      update_from,
      set_updated,
      ...
   ])).

% Get an attribute value
%
mv_component::get_attribute(Component,Attribute,Value) :-
   Component@Attribute(Value).

% Update attribute
%
mv_component::update_attribute(Component,Attribute,NewValu
e) :-
   default_value(Component,Attribute,nil,OldValue),
   Component@Attribute:=NewValue,
   Component@record_update(

   update_attribute(Component,Attribute,OldValue,NewValue)
,'Update Attribute'), !.

% Record update against component
%
mv_component::record_update(Component,Update,Name) :-
   Component@set_updated,
   Component@update_dependents(Dependents),
   mv_broadcast(Dependents,Update,Component).

mv_broadcast([],_,_) :- !.
mv_broadcast([Dependent|Dependents],Update,Component) :-
   Dependent@update_from(Update,Component),
 mv_broadcast(Dependents,Update,Component).

% Return all components dependent on changes to this
component
%
mv_component::update_dependents(Component,Dependents) :-
   default_value(Component,dependents,[],Dependents).

% Component has been updated
%
mv_component::set_updated(Component) :-
   Component@updated:=true.
```

```
% Process Update/AppUpdate from another component
%
mv_component::update_from(Component,Update,FromComponent).
```

fig. 7.1.   Part of the component class implementation in Snart.

Attributes are updated by calling `Comp@update_attribute(Attribute, NewValue)` for some component `Comp`. Update records are generated by calling `record_update(UpdateRecord, UpdateName)` with a Prolog term representing the update. Update records are broadcast to all dependents of a component (returned by `Comp@update_dependents(DependentsList)`) and these dependents interpret the update record with `update_from(UpdateRecord,FromComponent)`. The Snart framework provides more flexibility than MVSL for designating dependent components. Some relationships can be designated to relate dependents to a component (via over-riding of `update_dependents`) and some components can be made dependents dynamically (by calling `add_dependent(Component)` and `remove_dependent(Component)`).

Snart methods are executed in the same manner as Prolog predicates and several method implementations can be defined for the same method name. This supports an abstract implementation of `update_from` by defining an `update_from` method implementation for each kind of update record (and update record values) a component should respond to. This supports MVSL update operation selection (dependent on update record kind and number and type of values) via declarative Prolog "pattern-matching" using multiple method implementations for `update_from`. Fig. 7.2. shows an example component `update_from` method that responds to different update record kinds and values. The last `update_from` method implementation passes the update record to the `update_from` method for the parent class of `comp` for processing.

```
comp::update_from(Comp,update_attribute(Comp,name,OldName,
NewName),Comp) :-
   % i.e. update_attribute of name on Comp itself
   ...
comp::update_from(Comp,update_attribute(SubComp,name,Old,N
ew),SubComp) :-
   % i.e. update_attribute of name on a sub-component of
Comp, SubComp
   SubComp@comp_kind(comp2),
   ...
comp::update_from(Comp,delete_comp(SubComp),SubComp) :-
   % sub-component of Comp deleted
   ...
comp::update_from(Comp,establish(Rel,Parent,Comp),Rel) :-
   % relationship Rel established between Parent and Comp
   Rel@comp_kind(rel_comp),
   ...
comp::update_from(Comp,UpdateRecord,FromComponent) :-
   Comp@parent_update_from(UpdateRecord,FromComponent).
```

fig. 7.2.   Update operation implementation in the Snart framework.

`store_update` stores update records as terms but uses an additional method `app_update(UpdateRecord, AppUpdateRecord)` to convert an update record into an "application-specific" form. Any update records stored for a component must use the component's `unique_id`, rather than the component's Snart object reference. A `unique_id` must be used as stored update records are saved and reloaded and thus must not directly refer to Snart object ids which may change when components are reloaded. A declarative `get_update_text` method implemented for each `component` returns a list of atoms which are printed to describe the human-readable form of an update record. This is used for update record browsing and as the unparsed form of an update record in a textual display view.

In addition to the methods defined by the MViews architecture, `component` implements methods for list attribute management. List attributes are implemented as Prolog lists and can be used instead of one-to-many relationship components for very efficient one-to-many relationship implementation. A component must over-ride `establish_rel` and `dissolve_rel`, however, to manage such "list" relationships.

7.4.   Base Program Components

7.4.1.   View and Base Components

`view_comp` is implemented as a Snart class which inherits from component. `base_comp` inherits from `view_comp` and specialisations of `base_comp` (i.e. base components which are not viewable) are implemented by inheriting `base_comp`.

7.4.2. Viewable Base Components

`viewable_base_comp` uses a list attribute `subset_rels` for its subset/base relationships as object references. References to the subset views a base component owns are stored as a list of Prolog terms of the form `ViewName(Location)` where `ViewName` is the name of the view and `Location` is used to locate the view's persistent form. Subset views a base component is viewed in are stored as a list of subset view `unique_id` values. A dialogue allows programmers to browse and select the views a base component is represented in. These representations were used rather than relationship components to support very efficient subset view and relationship management.

7.4.3. Textual Base Components

Text form data for `text_base_comp` is stored as Prolog terms in a `component_text` list attribute. As with update records, storing text forms as objects is not necessary, as text forms only store data and are manipulated entirely by their owning base component[19]. A persistent storage location describes how to find the text associated with a text form and this text is displayed as the rendering of the base component in a textual display view.

7.5. Subset and Display Components

Fig. 7.3. shows the extended subset and display component hierarchies and methods.

---

[19]Hence using an object-oriented representation for text forms is not required. This flexibility of using Snart objects or Prolog terms to store data proved very useful for both MViews and IspelM. Typically, objects are used for MViews component data and terms for "structured" data associated with components (i.e. "complex" attribute values).
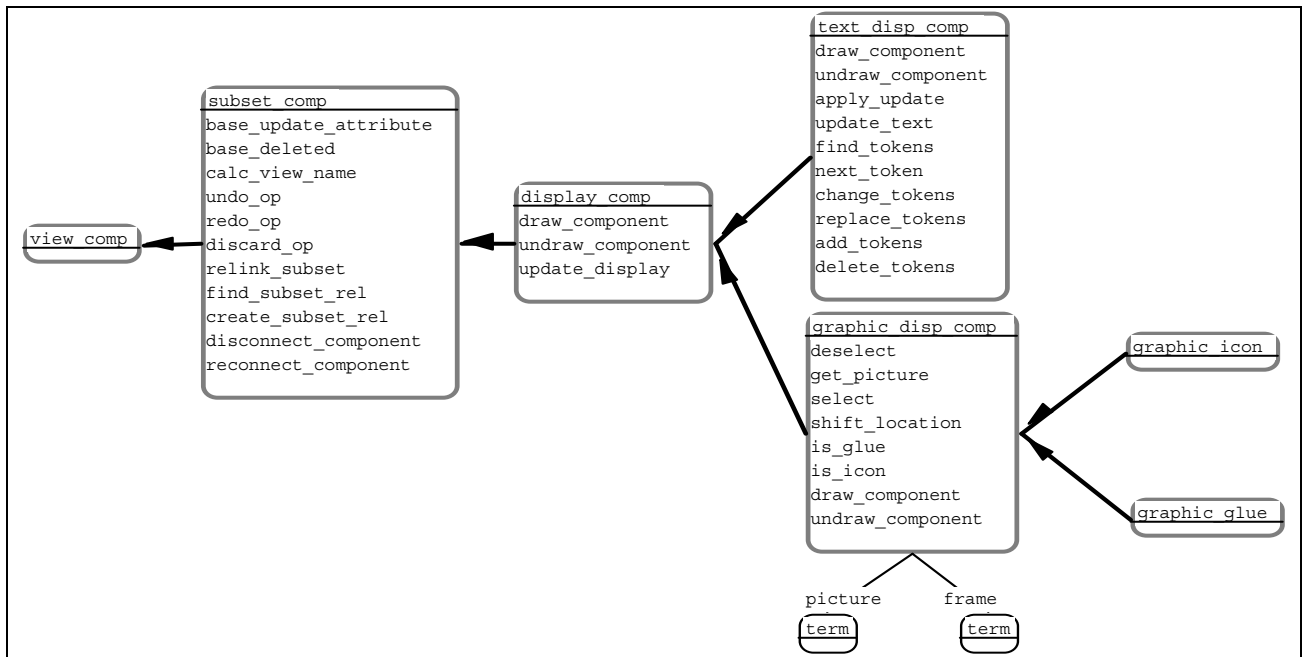
fig. 7.3.   Extended subset and display component classes for the Snart framework.

7.5.1. Subset Components

`subset_comp` defines `remap_on_reload` to remap a reloaded subset component to a base component possibly using the base component's stored update records to update its own state. The Snart framework permits in-core subset components only to be updated immediately when base component changes are made (for efficiency). `remap_on_reload` can be used to reconcile a reloaded subset component's state to the base.

`base_deleted` implements an efficient deletion method for subset components when their base view has been deleted (i.e. the current program closed by a programmer). `undo_op`, `redo_op` and `discard_op` implement declarative methods similar to `update_from` which determine how to reverse, reapply or discard an update record. `relink_subset` is used when a subset component is reloaded from persistent storage or copied using `copy` or `cut` to relink subset component relationships as new Snart objects are generated. `find_subset_rel` and `create_subset_rel` locate and create subset/base relationships for a subset component by their Snart class name. `disconnect_component` and `reconnect_component` remove and add a subset component to its enclosing subset view.

7.5.2. Display Components

`display_comp` uses a declarative form of `update_display` to determine if a display component should be re-rendered or not. Specialisations of `display_comp` can take further action on update records, for example only re-rendering part of their display (incremental updates) or unparsing an update record to indicate a change.

7.5.3. Graphical Display Components

Graphical display components are rendered as GDL pictures of arbitrary complexity. One display can be composed of several sub-display components which render different parts of a base and hence can be interacted with separately. These sub-displays are created and deleted by their "parent" and are dependents of their parent (hence they are notified of any changes their parent undergoes via `update_from` and can update their own state accordingly). The default action of `update_display` is to always re-draw the display component entirely (i.e. `undraw_component` and then `draw_component`). All `graphic_disp_comp` operations are implemented using GDL predicates with data stored in the `graphic_disp_comp` object.

When interactively selecting display component MViews needs to know if a mouse click or marqui selection has covered a component's picture. LPA's picture location predicate uses only the front picture item in a composite picture and this proved very unsatisfactory. MViews implements a selection mechanism using a frame defined by each class of graphical display component and determines whether the click-point is inside this frame or selection encloses the frame. Click-points within the frame can be defined using a list of terms of the form `Name(Top,Left,Height,Width)` stored in `click_regions`.

7.5.4. Textual Display Components

`draw_component` for `text_disp_comp` inserts the text of a base text form into a textual view's text window and `undraw_component` removes this text. This text processing uses LPA text window manipulation predicates. `update_display` called by the `text_disp_comp`'s subset component indicates base component update records are available for unparsing into the text. `update_text` is called to perform this unparsing (which uses the base component's declarative `get_update_text` method) and writes the token list returned into the text window.

Token processing for incremental application of updates is performed with LPA text window manipulation predicates and Prolog pattern-matching. These token manipulation methods include `find_tokens`, `add_tokens`, `replace_tokens` and `delete_tokens`. `find_tokens` incrementally parses each Prolog token in the view at a time and returns a match based on a given regular expression (or fails if a match can't be found). The matched tokens are returned as a list of terms of the form `(TokenStart,TokenEnd,TokenValue)` where `TokenStart` and `TokenEnd` delimit the token's range in the window and `TokenValue` is a Prolog atom describing the token's value. The returned token list can be used by other token manipulation functions to change the text in the view which thus applies the update record to the view. Fig. 7.4. shows an example `find_tokens` method call for a textual display view where a class definition start is being searched for (which may be text of the form "`abstract_class(Name`" or "`class(Name`").

fig. 7.4.

Textual view contents:

```
/* updates_start(78).
update(5). % rename window to gwindow
updates_end. */

class(window,
 parents([]),
 features([
  name:string,
  ...
 ])).
```

Prolog variables returned:

Name = window
NameToken=(78,83,window)

Method Call:

```
TextDisplayComp@find_tokens(
 [[abstract_class,'(', Name],
  [class,'(',Name]],
  [NameToken])
```
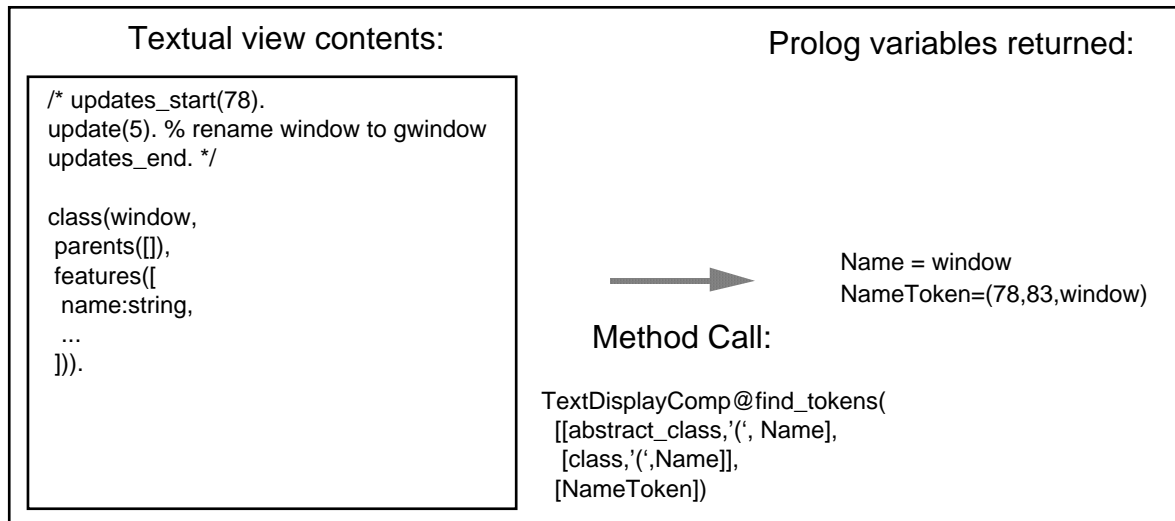
fig. 7.4.　　　An example of find_tokens for a textual display view.

7.6. Views

Fig. 7.5. shows extra methods and structure for views. The Snart framework view class provides methods to manage view component creation (`create_component(Kind,Component)`) and to translate a component's object reference to and from various forms (used for persistency management, schema evolution and framework specialisation). `component_to_kind(Component,Kind)` and `kind_to_component(Kind,Component)` translate a view component's Snart object id into its `comp_kind` value and vice-versa. If this kind has been renamed (i.e. an implementer of an MViews environment changes this value) these methods can be over-ridden to translate an old kind into an appropriate new `comp_kind` value.

Abstracting this component creation facility into view classes supports framework specialisation (for example, SPE specialised from IspelM) where specialised classes must create and manipulate classes from the same framework level as themselves. For example, SPE might define an `spe_program`, a specialisation of `program` from IspelM, which needs to manipulate `spe_base_class`, not `base_class` from IspelM (as `spe_base_class` extends `base_class` for Snart programming). `spe_program` can over-ride `kind_to_component` so any unspecialised IspelM classes used by SPE create the correct `spe_base_class` object.

Additional methods are provided to translate component objects to references (i.e. returns the Snart object id and its `comp_kind` which can be used to recreate and relink components with a list of terms of the form `OldRef(NewRef)`) and vice-versa using `component_to_ref(Component, Kind(Ref))` and `ref_to_component(Kind(Ref), OldRefList, NewRefList, Component)`. Similarly, `component_to_unique(Component, Kind(UniqueID))` returns a component's `unique_id/comp_kind` pair while

`unique_to_component(Kind(UniqueID), Component)` creates a component object given its `unique_id` and `comp_kind`.
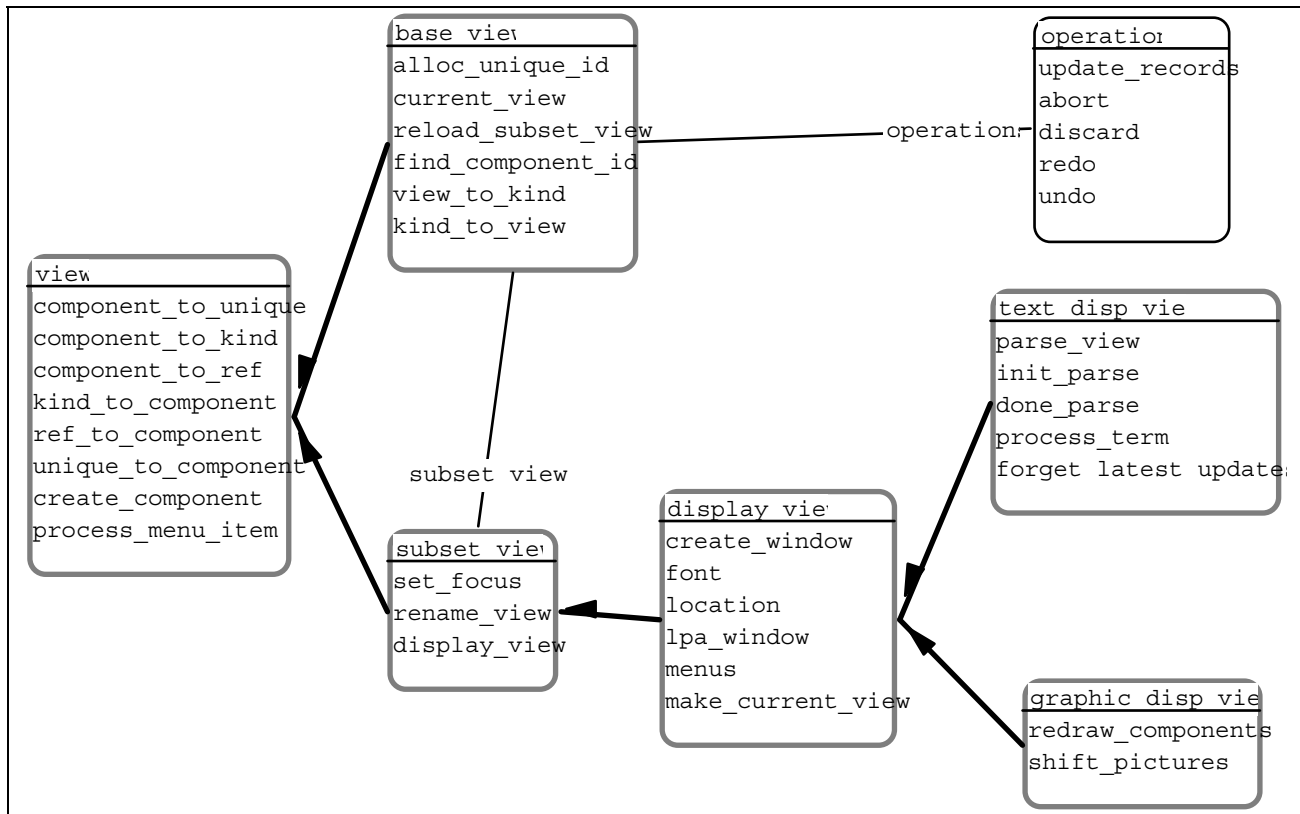
fig. 7.5.   Extra view methods and structure for the Snart framework.

7.6.1.   Base View

The base view records the `current_view` as an attribute and all subset view components currently in memory as a list attribute `subset_views` (for efficiency). Components can be located by their `unique_id` and loaded from persistent storage (for example, by calling `reload_subset_view`) when accessed, by `find_component_id`. The base view allocates values for `unique_id` methods with `alloc_unique_id`.

A base view stores a list of editing operations (as `operation` objects) using `operations` and these are used to provide a global undo/redo facility. An editing operation is composed of a list of update records that are sent back in sequence to their generating components for undo/redo. A base view also stores a list of subset views using `subset_views`. This allows all in-core subset views to be accessed from the base view.

7.6.2.   Subset Views

A renamed subset view must inform its display view of this change by calling `rename_view` so it can rename its LPA window. When the focus of a subset view is changed by `set_focus` the subset view may need to be deleted from one place on persistent storage and saved to another (depending on how subset views are stored).

When a subset view or one of its components is updated, any update records generated are recorded as a list of update record terms in an `operation` object, which the base view stores as a list attribute `operations`. After an `Undo` menu item is selected, the previous editing operation is reversed by reversing each of its component update records by calling `undo` for operation. Each update record is sent to its generating component for reversal (by calling methods which reverse the update record's change). A component may pass an update record to its parent class for reversal.

7.6.3. Display Views

Subset views are specialised to display views. All display views have an associated LPA window referred to by name using a string attribute `lpa_window`. Display views also provide additional support for window manipulation using `create_window`, `font` and `location`.

7.6.4. Textual Display Views

Textual display views use an LPA text window to display the text associated with base component text forms and LPA text window predicates are used for manipulating this text.

Parsing Support

`parse_view` is called to parse an updated textual display view. MViews assumes a Prolog syntax for text views and uses LPA window processing predicates to read terms and identify text forms. Each term read in is given to `process_term` which uses the textual display component that "owns" the term to compute changes in a base component. Methods are called for the base component to change its state to be consistent with the parsed text. Programmers are usually aware of the changes made to the text view and hence don't need to be informed of them via update records. Thus these base updates caused by parsing can be "forgotten" by a text component by calling `forget_latest_updates`.

The main complication with parsing involves identifying the text form which "owns" the term (i.e. the text form which encloses the term read in). Textual display views provide a `locate_component` method which, given the end position of the read-in term, locates the display component whose text form encloses the term. Originally the framework had `updates_start` as a term itself and the parser identified the "owning" display component

from its ID stored as `updates_start(ID)` [20]. This approach, however, means any existing language compiler (for example, the Snart compiler) can not read and compile the text window, as extra terms are present it doesn't understand. The current approach just adds updates as comments which are ignored by the standard Prolog term parser but are used by MViews textual display views and components.

Unparsing

Base information is unparsed into a text view when: a new textual display component is added to the view; a text form is generated when first displayed; or update records are applied on components in the view. Textual display components store the base text form id they render and this text form is generated by base components when required. A simple unparser provides predicates to write information based on a "template" into a given text window. This uses base component information to generate text and lays it out in an application-specific way. Garlan's flexible unparsing scheme (Garlan 86) and the unparsing grammars of the Mjølner environment (Minör 90) provide similar facilities based on unparsing languages for abstract syntax grammars.

Applying Updates

To have MViews apply an update to a view a programmer: selects the update records to apply using mouse selection; or asks for all updates records for the selected display components to be applied. MViews determines the update records to apply by reading their `update(Number)` part and then applies each update record in sequence to the view. If an update record can not be applied (either the textual display component does not implement an application for the update kind, or the view's text has been changed by the programmer so the update is no longer able to be applied[21]) its update record is left. If the update was applied successfully (i.e. the text changed), then its update record is removed from the view.

---

[20]In fact, the ID was originally unparsed as an application-specific name, for example "window" or "window::create". This approach means only one text form per base element can be displayed in a text view at one time. Thus a documentation text form could not be displayed with a code text form (hence the use of a textual display component ID now).

[21]IspelM can automatically apply such updates as renaming classes and features, adding or removing features from a class or deleting a class or feature from a view. Adding client-supplier relationships to a class can not be automatically applied (as they are implemented as feature calls) nor an update applied if the view's text has been altered (for example, a class or feature already renamed by the programmer and a rename update record is applied using its old name).

7.6.5. Graphical Display Views

Graphical display components may receive several update records from their subsets that result from one editing operation. For example, if a client-supplier glue component has two or more attributes changed via a dialogue, it will receive an `update_attribute` update record for each attribute change. These graphical display components need only be re-rendered once, however, for efficiency.

Graphical display views store a list of their components which require redrawing using a `redraw_components` list attribute. Graphical display components are entered in this list by `add_redraw_component(Component)` and are then marked as "being redrawn" by setting a boolean flag `redraw` for the component. A graphical object may be entered in the `redraw_components` list several times before is actually redrawn.

Graphic display views provide tools for manipulating display components. Each tool is implemented as an LPA graphics window tool which calls a graphic view method when selected, deselected, or there is a click in the window. MViews uses LPA predicates to implement mouse processing, dragging of pictures, marqui selection, rubber-banding and text editing and graphical display views provide these facilities as methods.

Copying and pasting graphical objects is more complex than copying text. Selected display components are duplicated and references updated to the copied objects (using `component_to_ref` from the display view). When pasting objects, the copied components must again be duplicated and then added to the new view, their references updated, and then be redrawn. The MViews framework does not currently support the copying of base data in this manner, but calls `remap` on the pasted components to relink them to the base or to recreate base data.

7.7. Relationships

7.7.1. Relationship Classes

Relationship components are modelled by relationship and its specialisations. They can also be modelled by component or component list attributes, if desired. The advantage with modelling relationships as component classes is that they support component methods, and can hence be referred to directly, made dependents of other components, and be dynamically purged and reloaded from persistent storage. Multiple inheritance between Snart classes is used to define base and subset components as base and subset/base relationships.

7.7.2.   Subset/base relationships

The `subset_rel` class translates subset component update records into operations on their base component(s) if the base components are affected by the subset component updates. It also translates base component update records into subset component operations if the subset components are affected by the base updates. Updating a component attribute, establishing or dissolving component relationships, and creation and deletion of a component are all operations that may need to be propagated between a base component and its subset components.

As a base component may have several subset components in several views. As updating a subset component and re-rendering its display component is often costly, only subset components in the current (front) view need be immediately updated. If a subset/base relationship receives update records from a base component and one or more of its subset components are not in the current view, the update records received are stored against the subset components' views until they become the current view. Then any components with updates are modified in the normal way[22]. Subset views record a list of update records and affected subset components to implement this process.

Currently the default subset_rel class propagates updates to in-core subset components. Reloaded subset components are reconciled to their base component state using `remap_on_reload`. This is used so MViews systems can support many subset components for one base component with little impact on interactive performance when updating these subset components. Sub-classing `subset_rel` can over-ride this default behaviour if necessary.

`subset_rel` supports semi-automatic propagation of base and subset component attribute updates using a `base_to_subset` method. `base_to_subset` can be redefined in specialisations of `subset_rel` to define base component to subset component attribute mappings. `base_to_subset` returns a list of terms of the form `BaseAttribute(SubsetAttribute)` which is assumed to map base component attributes to their subset component equivalent. For example, `base_class` from IspelM might define `class_name` and `kind` and so might `class_icon`, hence a `class_icon_subset_rel` mapping of `[class_name(class_name), kind(kind)]`. The default behaviour of `subset_rel` is to

---

[22]The updates could be processed in the background using idle time, similar to the incremental attribute recalculation schemes of [Reps and Teitelbaum 87] and [Hudson 91]. LPA currently does not provide such idle time processing facilities so MViews processes these delayed updates "on-demand" (when an updated view is made the current view).

translate between these base and subset component attribute updates automatically, but this can be over-ridden in sub-classes to support different attribute mappings.

7.8. Operations and Updates

7.8.1. Update Record Generation and Storage

The `record_update(UpdateRecord,UpdateName)` method is given update records as Prolog terms and a human-readable name describing the update as an atom. `store_update(UpdateRecord)` numbers each stored update record sequentially and stores these update records as terms. `store_and_record_update` stores an update record and also propagates it to the storing component's dependents. This is useful for sub-components that do not store update records themselves.

7.8.2. Update Record Propagation

Subset views implement undo and redo by recording a sequence of update records in an `operation` object and the base view maintains a list of these operations which forms an "editing history". An editing history browser dialogue is provided by MViews which allows several updates to be undone or redone at one time. This is implemented by displaying a menu of operation names and allowing a programmer to select an update record to undo or redo up to.

Update records are unparsed and printed in textual display views and an update record browser dialogue as required. User-defined update records of the form `user_update(Tokens)` can be added arbitrarily to document changes at a user-defined level of abstraction. Extra comments can be associated with update records via the updates browser and these are stored as a list of atoms. Update records are deleted by the update record browser on user request or by MViews components (for example, when semantic errors corrected or a new class compilation in IspelM performed). Update records can also be moved to a component's "update history". These history update records are not shown unless a programmer specifically asks to view them[23]. This improves efficiency when a base component stores many update records and allows programmers to view only recent update records they are currently interested in.

---

[23]Which leads to a concept of "active" update records associated with an element and "history" update records documenting old changes to the component.

7.8.3. Constraints and Semantic Calculation via Operations

Sub-classing allows constraints and semantic calculation to be associated with operation methods. In IspelM, a feature sub-class of an MViews base class can implement operations for renaming the feature, attaching the feature to a class and changing the feature's type. The methods for these operations can include Snart code which ensure, for example, features of a class have a unique name and the type for a feature is valid (either one of the pre-defined Snart types or a class name).

7.8.4. Lazy Application of Update Records

The Snart framework provides support for lazily processing update records. On receipt of an update record a component can record it in a list of update records stored in `lazy_updates`. These update records can be processed together at a later date by calling `apply_lazy_updates` for the component. This is useful for generating composite update records which reflect more than one basic operation on a component and for determining update record responses which depend on more than one update record.

For example, one approach to generating a `shift_location(DX,DY)` update record for graphical display components might be to store `update_attribute` update records for the `x` and `y` attributes of the component. After the editing operation on the graphical display view has finished MViews can call `apply_lazy_updates` for any updated graphical display components. If update records of the form `update_attribute(Comp,x,OldX,NewX)` and `update_attribute(Comp,y,OldY,NewY)` are in the `lazy_updates` list for the component, a new update record `shift_location(DX,DY)` (where `DX=OldX-NewX` and `DY=OldY-NewY`) can be generated and propagated and the two update_attribute records discarded.

Care needs to be taken when using lazy update application and propagation that mutually dependent components receive update records at appropriate times (otherwise their states may be incompatible until all lazy update records have been processed). Chapter 10 discusses enhancements to MViews which would provide improved support for lazy update processing.

7.9. User Interaction

Users interact with MViews programs via display views, with each display view having its information rendered inside an LPA window. The front LPA window denotes the current view and all editing operations are applied to this view.

In addition to display views, programmers interact with MViews through menus and dialogues. Menus provide a structured mechanism to apply operations to display, subset or base views. Display views interpret LPA menu selection events sent to them using a declarative `process_menu_item(MenuName,ItemName)` method. `process_menu_item` method

decodes the menu selection and calls appropriate methods for the display view or selected display view components. If the display view does not handle the selection, `process_menu_item` calls `process_menu_item` method for its subset view (renamed by Snart), which in turn may call `process_menu_item` for its base view or application. This menu selection propagation allows menu processing to be handled by the most appropriate view. LPA menus for different kinds of display views can be enabled and disabled by calling `menus`, called when the display view's subset view becomes the current view (see Appendix A).

An LPA dialogue is comprised of interactors including text fields, editable text fields, radios and check boxes, menu selections, and buttons. Dialogues are defined using LPA dialogue manipulation predicates (see Appendix A). Dialogues either return entered information, display information they are given, or update component objects directly using method calls.

7.10. Persistent Program Storage

Programs must exist from one invocation of an MViews environment to another. The MViews architecture assumes no special method of storing program data but the Snart framework provides three approaches of differing levels of abstraction.

7.10.1. Term Data Files

The lowest level of persistency management uses "term data files" which provide a basic mechanism for saving and loading Prolog terms to Macintosh resource files. Various predicates are provided which create, open and close term data files and read, write and delete terms in data files using resource ids.

LPA does not provide any Prolog term read and write facilities for resource files but provides atom read and write facilities (i.e. text sequences up to 255 characters). MViews extends this facility so terms can be read and written to resource files by:
- writing a term to an LPA text window
- saving the window as a sequence of atom resources
- reading a sequence of atom resources into a text window
- reading a term from the text window

Resource files are used rather than reading and writing of Prolog terms to text files for efficiency and so random access using a resource id can be used. This allows many terms to be saved in the same file and hence increases efficiency (less files need to be opened) and terms can be read, written and deleted in any order. The Macintosh resource manager maintains the resources and handles garbage-collection and resource file compaction.

7.10.2. Component Persistency Methods

Direct use of term files is not very abstract so MViews augments the `component` hierarchy with extra methods to save and reload components. Components provide a `save_data` method which returns all their data that needs to be made persistent as Prolog terms and a `load_data` method that rebuilds the component from its persistent data on reload. `save` and `load` are called to write and read a component (and possibly its sub-components) to and from persistent storage. Semantic attribute values can be saved in the same manner as program structure (as they are stored in the same form) or can be recalculated when a component is reloaded. Relationships are saved as components or as attributes (if represented as Snart object attributes or list attributes). `save` and `load` use the term data file predicates and a resource id to save and reload component data.

If Snart object references are used to relate components these must be saved in some persistent form and object references re-established on reload. The `component_to_ref` and `component_to_unique` methods associated with views and the base view's unique id look-up tables support this relinking process. A unique id typically stores the component's type and a unique id number allocated by the base view for every component. It can also contain the unique id for any parent component needed to locate the component. For example, the unique id for a feature in IspelM is of the form `feature(ClassID,FeatureID)`.

An MViews program need only be partially in memory at one time. Only some related components need be in-core and components can be purged (written to persistent store if updated and then deleted from memory). This allows some of a program's views to be cached and some of the program to be loaded. An MViews environment must ensure appropriate view and program information is reloaded when required and this is currently assumed to be managed in an application-specific manner (either by reloading a component when accessed or reloading groups of sub-components for a component).

Some MViews components can be partially in-core (i.e. have only some of their attributes and relationships in-core) and have only updated information saved. This incremental saving and loading of component data is supported by augmenting the base component operations with *group* management operations. Groups are identified by name and can be saved and load independently.

MViews allows for schema evolution when an environment's program storage is modified or extended as new tools or facilities are added. The save and load methods can use a version number to identify the format of reloaded data and take different restore actions for different versions of a component's structure.

7.10.3. Snart Object Persistency

A third method of storing component data uses the Snart object persistency mechanism described in Chapter 3. This is very language-specific as it assumes the implementation language is a "persistent language" with objects saved and reloaded when required. This is the most abstract approach to program persistency but the limitations of the current Snart object persistency mechanism mean only one program can be open at a time (as multiple object stores are not currently supported).

7.11. Discussion and Future Research

7.11.1. Component Class Implementation

As the MViews architectural model is object-oriented it translates well into a framework of Snart classes. Storing component attributes as Snart object attributes works well and has proved an efficient way to implement the component attributes used by the MViews architecture. Relationships as object references and lists of object references also provide an efficient way to implement one-to-one and one-to-many relationships. The main disadvantage with using attributes to model relationships is that `establish_rel` and `dissolve_rel` methods must be implemented to manage them and generate appropriate update records. One solution to this problem might be to define extra methods for relationship attribute and list attribute processing which generate `establish` and `dissolve` update records.

Environment program representations are implemented by sub-classing Snart component classes and defining appropriate attributes, relationships and methods to store data, relate components, and provide specialised component operations. This works well for defining the structure and some semantic relationships between components of a program. More complex semantics, in particular the behaviour of programs as opposed to static constraints, are not so easy to implement in this framework. Combining MViews program structure storage with an attribute grammar or similar semantics specification approach may alleviate these problems. This is the approach taken by the SByS structure-oriented editor (Minör 90) and the approach assumed by Kaiser's attribute grammars (Kaiser 85).

While it is reasonably easy to specialise the Snart framework to implement an environment, a declarative specification language like MVSL may still be useful. Such a language could be used to generate MViews framework classes which could then be further specialised to express operations and data in ways not easily done by MVSL or for reasons of efficiency.

7.11.2. Operations and Update Records

Implementing the MViews architecture class methods as Snart methods was a natural approach to providing basic operation methods and component-specific operation methods. The use of a declarative `update_from` method achieves a level of abstraction close to that of MVSL's update operations for defining a component's response to update records.

Implementing update records as Prolog terms proved an efficient and flexible approach. Originally these were implemented as objects, but this proved slow and cumbersome. Classes had to be defined for each kind of update record (thus forming a hierarchy), objects created and initialised for every update record then propagated by calling `record_update`, and matching an update record object could not use as abstract a form of `update_form`. This process caused a large performance over-head. As we used update records we determined that associating functionality with them (i.e. an object-oriented approach to storing update records) was not particularly useful. Most often functionality was dependent on the kind of component using the update record, not the update record kind. Thus components now implement update record creation (by calling `record_update`), reversal (`undo_op`), redoing (`redo_op`), and discarding (`discard_op`). Sub-classing allows the behaviour of update record treatment to be modified for specialisations of a component class.

MViews assumes environments treat undo as a history of reversible (and redoable) update records. If undo is to be treated as an editing operation itself (i.e. an undo is undone by another `Undo` operation), sub-classing the base view can provide this. Allowing updates to be undone and redone in an arbitrary order, and allowing a group of updates to be applied to a generic component to implement macros, would be useful. To implement such facilities, each component's update record `undo_op` and `redo_op` methods would need to ensure such an operation is valid before performing it. Such a system would need to provide some mechanism of informing programmers of "invalid" updates, i.e. updates that couldn't be undone/redone as they no longer make sense (due to undo/redo of previous or subsequent updates). Chapter 10 discusses this further in the context of version control using update records.

7.11.3. User Interaction

Dialogues and Menus

Interaction with MViews environments is via display views or menus and dialogues. Using LPA dialogue and menu predicates directly works well but is a problem when dialogues and menus need to be specialised. Specification using absolute screen co-ordinates is also not as abstract as an interface builder supporting interactive dialogue

construction (see Chapter 9). Provision for defaulting field values and checking entries after editing dialogue fields would be very useful. A better correlation between display views and dialogues may also be useful (i.e. treating dialogues as display views with dialogue-style interactors).

Textual Display Views

Textual display views provide a novel method of integrating free-edited textual program detail with interactively-edited, high-level program data. Textual views are assumed to have a Prolog-readable format and hence parsing uses a Prolog-supplied `read` predicate on text windows. Unparsing is done either by generating a text form (when creating a new text form) or incrementally on a text form when updates are applied to a view. This parsing/unparsing support is sufficient for systems using a Prolog-based textual view syntax but needs substantial enhancement to support more general parsing (possibly using Definite Clause Grammars (LPA 89a) or a yacc-like parser generator front-end).

As noted in Chapter 4, textual interaction is simplistic with textual displays providing basic text editing operations and menus supporting access to subset and display information. A text editor incorporating both structure-oriented and free-editing modes, similar to the UQ2 editor (Welsh et al 91), would allow a more natural and useful editing of high-level program structure. Currently unparsing and parsing are disjoint activities, as are generating a new text form on creation (for example, a class definition's text) and automatically applying updates to a text view (by modifying its text). A closer relationship between these two forms of unparsing and parsing would make specification of textual views easier and more extensible (for example, all based on a grammar). Extending textual display views to support hyper-text links for fast navigation and structure-oriented editing may be useful but would necessitate a much more sophisticated treatment of text windows. Fine-grained textual forms are supported in a limited way but improved textual annotation capabilities would enhance this support[24].

Using an existing text editor rather than the built-in LPA text editor would provide a more extensible environment with users being able to select their preferred editor. This approach, however, would not allow users to expand other text forms into a view or selectively apply updates as easily. A similar facility could be built using editors such as vi or emacs by writing interface code between the editor and MViews to perform the changes

---

[24]Fine-grained textual support means allowing parts of a term to be linked to different base components (i.e. more than one "updates_start" link for each term). This would be useful for filtering out more term information and also for multiple views of parts of a term (for example, for supporting views of pre- and post-conditions, similar to Eiffel [Meyer 88, 92]).

on the views' text (for example, using shell-scripts or C code). The user interface provided, however, would not be as seamless as that of the current environment.

Graphical Display Views

Graphical displays could be extended to support a Unidraw-like diagram editor model whose generic facilities would allow a wider range of graphical editors to be constructed more abstractly than at present. Automatic layout of graphs (Tammassia et al 88, Paulisch and Tichy 90, Mannucci et al 89), scalable glue and connector pins, and "parsing" of graphical representations (Golin and Reiss, 90) would all enhance the power of MViews graphical display views.

Browsing and Complexity Management

One aspect of the MViews framework that requires further enhancement is its support for browsing and complexity management. This can be built out of dialogues and display views, as has been done for IspelM, but little support is given directly at the MViews level. Implementing such capabilities have proved very important to the useability of SPE and hence more appropriate building blocks should be provided at the MViews level. This could include generic classes or predicates that implement menu dialogues for component browsing, support for partial base component viewing at the subset and display component levels, and filtering mechanisms (active constraints) based on component attribute values (useful for class responsibilities for IspelM).

7.11.4. Persistent Program Storage

One problem with the Snart framework is its handling of program persistency. Experience with developing IspelM has shown that using `save` and `load` methods associated with component classes is a less than ideal mechanism for storing programs. The disadvantages of this approach include:

- Difficulty in implementing specialised component save/load operations. Ensuring that all required data is in-core for a component is often quite difficult as is relinking Snart object references using unique id values. For example, when mapping a subset view-level feature to a class IspelM must ensure the class is in-core and the required feature is in-core. No facilities are currently provided by the framework to automatically reload components when accessed.

- Coarse-grained saving is used where data is converted from Snart object form to savable data (using `save_data`). While this allows for a declarative reload predicate, thus supporting schema evolution, extending an environment to support new tools using the same base data is complicated. The base components must reload the saved data in the same form as stored, thus a tool not requiring some data still must reload it all.

- Look-up tables and relationships must be converted to and from persistent forms. This process should be automatic but if Snart object references are used these must be converted to and from a persistent form.
- Shared program data and version control are not supported and only one MViews environment can work on a program at a time. This makes multi-user development impossible and limits the size of systems that can be built.

Programs developed using Dora are stored in memory as C++ objects and are saved in a PCTE object store using database commands (Wang et al 92). GARDEN uses an object-oriented database management system which also supports transaction processing (Reiss 86). GARDEN objects are migrated to persistent storage as database objects and reloaded as in-core objects for efficiency. Unidraw writes internal diagram components (stored as C++ objects) to text files in a catalogue. EDGE (Newbury and Tircher 90) and Dannenburg's list system (Dannenburg 90) assume a text file representation scheme which is parsed to reload data. Persistent languages treat run-time entities as persistent objects which survive beyond one execution of a program with completely transparent saving and reloading of data (Sajeev and Hurst, 92). A combination of these approaches may be useful for MViews program persistency (see Chapter 10).

The Snart framework does not currently support different versions of the same program, as do the Mjølner environments (Magnusson et al 90, Minör 90), nor does it provide selective base program views for different tools as do MELD (Kaiser and Garlan 87) and Dora (Wang et al 92). Version control, configuration management and multi-user program updates are not supported. Chapter 10 discusses extending the Snart framework to provide version control using update records, configuration management, and distributed multi-user programming.

7.11.5. Implementation Language

Snart proved to be a good implementation language for MViews. Object-oriented support including very flexible multiple inheritance was almost essential for developing the framework. Integration with LPA predicates provided a large amount of ready-built support, particularly for graphics and user interface building, which greatly enhanced the development process. Declarative predicates for update operation processing (`update_from`), undo and redo of update records (`undo_op` etc.), converting update records to different forms (`app_update` and `get_update_text`), and applying update records (`apply_update`) greatly simplified the implementation of MViews and its derivatives. Backtracking via predicate failure proved useful for operation abortion and some constraints checking. As some implementation of the MViews and IspelM frameworks was experimental programming, with changes to classes and the model being frequent, the choice of Snart for implementation proved worthwhile.

MViews could be implemented using a different Prolog (for example, Quintus with ProTALK (Quintus 91b) or LPA with Prolog++ (Pountain 90)) or similar experimental programming language such as CLOS or Smalltalk (Goldberg and Robson 84). It could also be implemented in a strongly-typed object-oriented language such as Eiffel or C++. Attributes would need to be stored as objects and their value types checked at run-time, as done with an earlier version of MViews implemented in THINK C. This would allow attribute names to be determined at run-time for methods like `update_attribute`.

The main disadvantage with strongly-typed languages is that support for the declarative aspects of MViews is not directly provided. It is also much more difficult to modify the hierarchy or method arguments (as many classes must be recompiled) in an experimental way. As the MViews and IspelM models and frameworks have reached a point of some stability, however, it is unlikely such major changes as occurred during their development are as likely if implementation in another language is performed. The improved performance from a strongly-typed language would be of great benefit for developing larger software systems.

Update records can be implemented in other languages as either objects, record-style structures, or "terms" stored as lists of values. While most other languages do not provide Prolog's unification-style pattern-matching, a similar processing of update records can be done using case statements or by implementing a `unify` function.

7.12. Summary

MViews has been implemented as a framework of Snart classes. This object-oriented implementation supports the MViews architecture's abstractions and allows new environments to be developed by appropriate specialisation of this framework. Basic class groups include base components for program representation, subset and display components for viewing and rendering part of a base program component, and views for grouping program graphs and interactively modifying program renderings. Component attributes and some relationships are stored as Snart object attributes and operations implemented by methods. Update records are stored as Prolog terms and are interpreted by declarative methods. Additional support includes saving and loading of data to Macintosh resource files, declaratively specified dialogues and menus using LPA, parsing and unparsing of text as Prolog terms, and generic textual and graphical program manipulation methods.

Novel aspects of this implementation include the treatment of update records as Prolog terms generated and processed in a declarative style. Update record terms are used to document change (stored as application-specific terms), ensure view consistency and provide object dependency propagation, describe textual form updates in a human-readable form (update records are used to generate a Prolog atom list which is printed),

and support a generic undo/redo facility (by recording a list of component object update record terms). MViews' object dependency scheme also provides a mechanism for supporting data-driven and lazy attribute recalculation.

Many extensions to the Snart framework are possible. These include more abstract component persistency with shared access to programs, improved textual and graphical editor construction facilities, and extended support for update records and language semantic calculation. This framework is sufficient to demonstrate that the MViews architecture is realisable by an implementation, and that MViews abstractions can be reused to implement environments supporting multiple textual and graphical views with consistency management. Chapter 8 describes a model for IspelM using the MViews architecture and an implementation of IspelM and SPE using the Snart framework. Chapter 9 illustrates how both the MViews architecture and its implementation can be reused to extend IspelM and SPE and to construct other environments which support integrated textual and graphical views of information.

Chapter 8

Architecture and Implementation of IspelM and SPE

Chapter 4 described the user's perspective of the Snart Programming Environment. Chapter 5 discussed how the common aspects of multiple textual and graphical views with consistency could be factored out of SPE and similar environments to produce the MViews model. Chapter 6 used this model as the basis for an object-oriented architecture for designing such environments by reusing the MViews model. Chapter 7 presented a Snart framework implementing the architecture of Chapter 6 and thus demonstrated that MViews is realisable by an implementation.

In this Chapter we demonstrate that the major aspects of SPE can be factored out into IspelM to produce a generic environment for programming object-oriented languages. An object-oriented architecture for IspelM is developed by specialising classes from the MViews architecture. This model describes how object-oriented programs are stored, the different views of a program provided, support for browsing and complexity management, and incremental saving and loading of programs. An implementation of IspelM using the Snart framework from Chapter 7 is briefly discussed. The IspelM implementation is further specialised to support Snart programming including integration with the existing Snart compiler and run-time system (described in Chapter 3). The IspelM architecture, its implementation, and the SPE implementation are evaluated and future extensions proposed.

8.1. IspelM Architecture

Chapter 5 introduced IspelM, a generic software development environment for object-oriented languages supporting multiple textual and graphical views of an object-oriented program. The MVSL specification for IspelM defined base class and feature elements and base generalisation and client-supplier relationship components. Subset views and components included class diagram and class code views, class icon and feature text elements, and generalisation and client-supplier glue relationships. MVisual defined the display view renderings and interaction mechanisms used by IspelM.

As Chapter 5 noted, there is not sufficient information in the MVSL and MVisual specifications to automatically generate an implementation for IspelM. To design an implementation for IspelM the object-oriented architecture described in Chapter 6 can be reused. This design is based on the MVSL and MVisual specifications in Chapter 5 and Appendices D and E.

8.1.1. Overview of the IspelM Architecture

IspelM's components can be described as specialisations of different classes from the MViews architecture. Fig. 8.1. illustrates the basic specialisations used to describe IspelM.
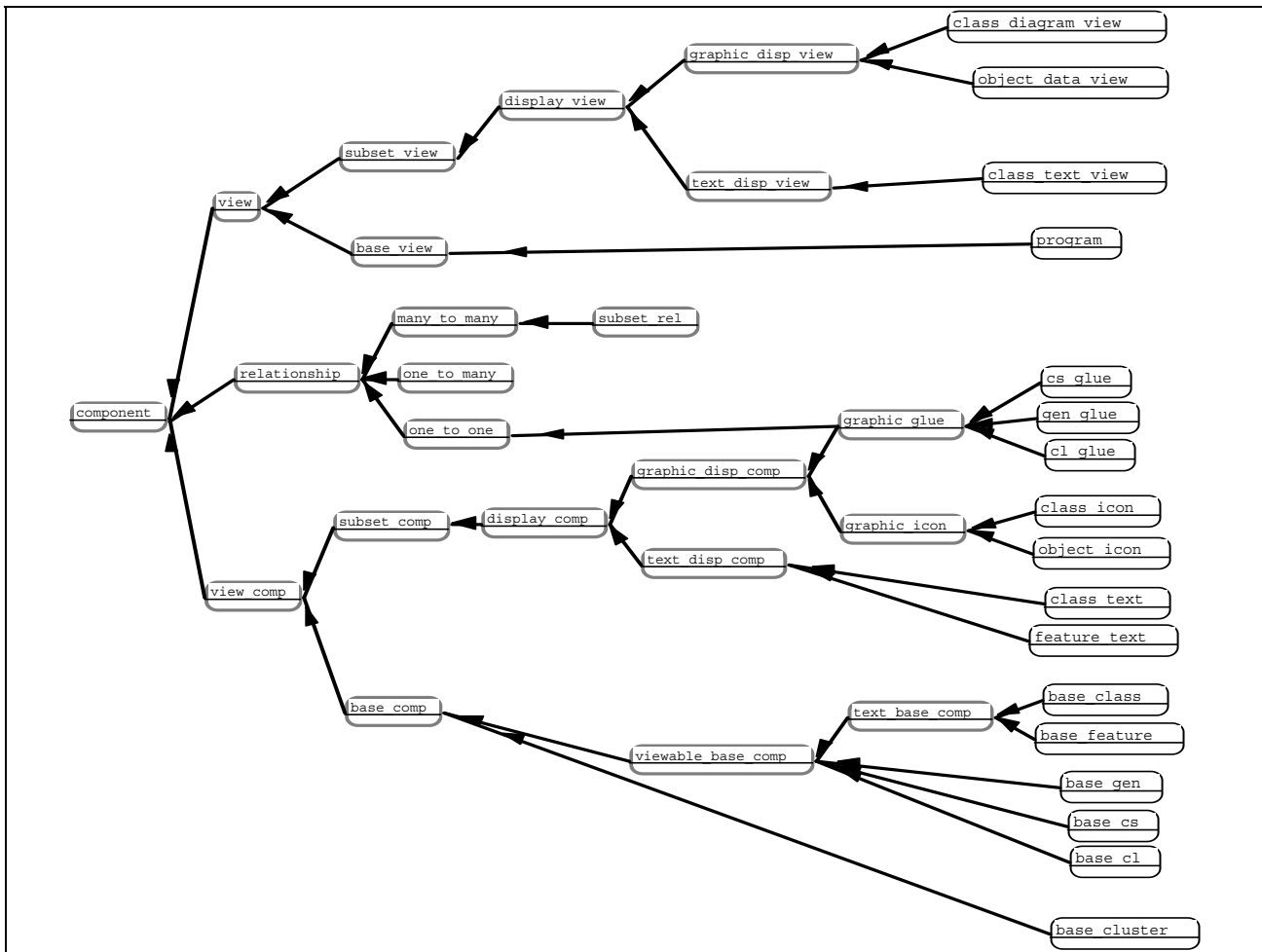
fig. 8.1. The basic component classes for IspelM.

In fig 8.1., the MViews classes are the abstract classes defined by the MViews architecture. The IspelM classes are shown on the far right as specialisations of different MViews classes. As the MViews architecture is used to model MVSL and MVisual components as classes, appropriate specialisation of these classes allows IspelM to model its components as classes. Also defined by IspelM, but not shown in fig. 8.1., are various subset/base relationship components, which are specialised from `subset_rel`. The IspelM classes are described in the following sections.

8.1.2. Base Clusters

Clusters are used to group classes according to a common purpose and are introduced by the IspelM architecture. For example, if IspelM were to store an SPE implementation, cluster groups may include "MViews classes", "IspelM classes", "SPE classes", and "Misc. classes". Clusters are not currently viewable in IspelM and thus `base_cluster` is described as a specialisation of `base_comp`. Fig. 8.2. shows the basic cluster structure and operations.
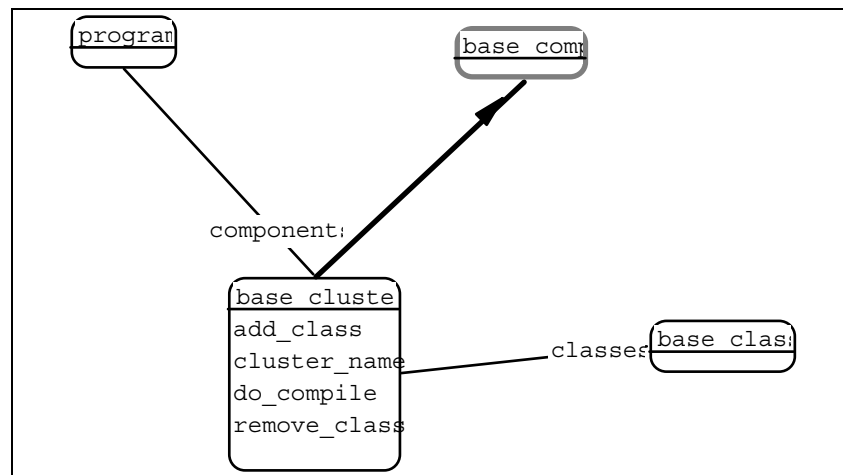
fig. 8.2.   Base cluster class structure and methods.

8.1.3. Base Classes

Base Class Components

IspelM defines base class elements to store information about each class of object for an object-oriented system. IspelM models MVSL base class elements as a base component class `base_class`. Base classes can have subset components in subset views and can have textual forms. Thus `base_class` is defined to be a specialisation of `text_base_comp` from the MViews architecture. `text_base_comp` supplies structure and methods to support subset views, subset components and text forms for `base_class`.

Fig 8.3. shows the structure of `base_class` and the methods supplied for manipulating class data, components, interface compilation, and views. This diagram groups related attributes, relationships, and methods in generalisation "classes" for `base_class` to reduce the cognitive complexity of the diagram.

text_base_com

class_element
add_cl
add_cs
add_feature
add_gen
find_cl
find_cs
find_feature
find_gen
remove_cl
remove_cs
remove_feature
remove_gen
establish_rel
dissolve_rel

class compil
set_updated
needs_compiling
do_compile
compile_error

class data
update_attribute
class_name
kind

class view
view_name

features
base feature
classifier
base cl
gens
base gen

all feature
all feature
kind
name
owner_name
owning_class
type

client supplier
specs
base cs
base class

base class

fig. 8.3.　Base class structure and operations.

Base Class State

Base class attributes and relationships defined by MVSL are modelled as class attributes and relationship components in the IspelM architecture. Structural class information includes a one-to-many relationship to the features owned by the class as features, which corresponds to the MVSL declaration features : one-to-may feature from Chapter 5. Other relationships and attributes include: generalisation relationships to parents of the class as gens to base_gen component relationships (i.e. gens : generalisation.child); the class name (class_name) and its kind (kind) as attributes; a one-to-many relationship to specialisation classes as specs (i.e. specs : one-to-many class); client-supplier relationships to associated classes as client_suppliers to base_cs relationships (i.e. client_suppliers : client_supplier.parent), and the complete class interface (i.e. all inherited and owned features for the class) as all_features to all_feature components (i.e. all_features : one-to-many all_feature). An additional kind of class-to-class relationship is the *classifier,* used to define how a class can be classified to its sub-classes. This can be used to model classification for languages such as Kea (Hosking et al 90) and Snart (see Chapter 3)) or used for analysis in such systems as OMT (Rumbaugh et al 91).

Base Class Component-Specific Operations

Structure management operations defined by MVSL are modelled as methods for `base_class`. These methods support initialising new components and relationships and establishing relationships to them by over-riding `establish_rel` and defining `add_feature`, `add_cs`, etc. These methods correspond to the `add_cs` and `add_feature` operations from the MSVL specification for base class elements which use basic component and relationship operations. The IspelM architecture methods reuse methods defined by classes from the MViews architecture. For example, `Comp@update_attribute(Attribute, New)` is used for the MVSL operation `Comp.Attribute := New` and `Parent@establish_rel(Kind, Parent, Child, NewRel)` for `establish(Kind, Parent, Child, NewRel)`.

Additional base class structure methods include removing relationships (`dissolve_rel`) and locating class components (`find_feature`, `find_cs`, etc.). Class components can be located by name (features and classifiers), information about the component (generalisations and client-suppliers), or by their `unique_id` value. Classes also provide methods for generating class interface information (i.e. compiling a class) as `do_compile`. Compilation errors are stored as update records against the class and are deleted at the start of a class compilation.

Base Class Update Operations

MVSL defines an update operation for base classes to ensure the class name is unique for a program. This can be implemented in two ways for the IspelM architecture. `update_from` can be redefined for `base_class` to check update records of the form `update_attribute(Class, class_name, OldName, NewName)` and generate semantic error update records if a class rename is invalid. Alternately, a constraint can be defined by over-riding `update_attribute` inherited from `text_base_comp` (and defined by `component`) to check for this problem.

8.1.4. Class Components

MVSL defines base elements and relationships for class sub-elements and relationships. The IspelM architecture models these as components and relationship components by defining component classes to model these MVSL definitions. Fig. 8.4. shows the structure and methods of class components and relationship components and the MViews classes they inherit from.

Features

The MVSL base feature element is implemented by the IspelM `base_feature` class which is a specialisation of the MViews `text_base_comp` class. Features can have subset

components and text forms and are thus specialised from `text_base_comp` as are base classes.

`base_feature` uses attributes to store information about its owning class as `owning_class`; its name, which is unique to its owning class, as `feature_name`; its type as `type_name`; and its kind (attribute, method, deferred, or inherited) as `kind`. `update_attribute` is redefined to ensure feature renames are valid and `view_name` computes a name for subset views of a feature. Base features record updates by sending them to their base class and may then store the update themselves. This is accomplished by redefining `record_update`.
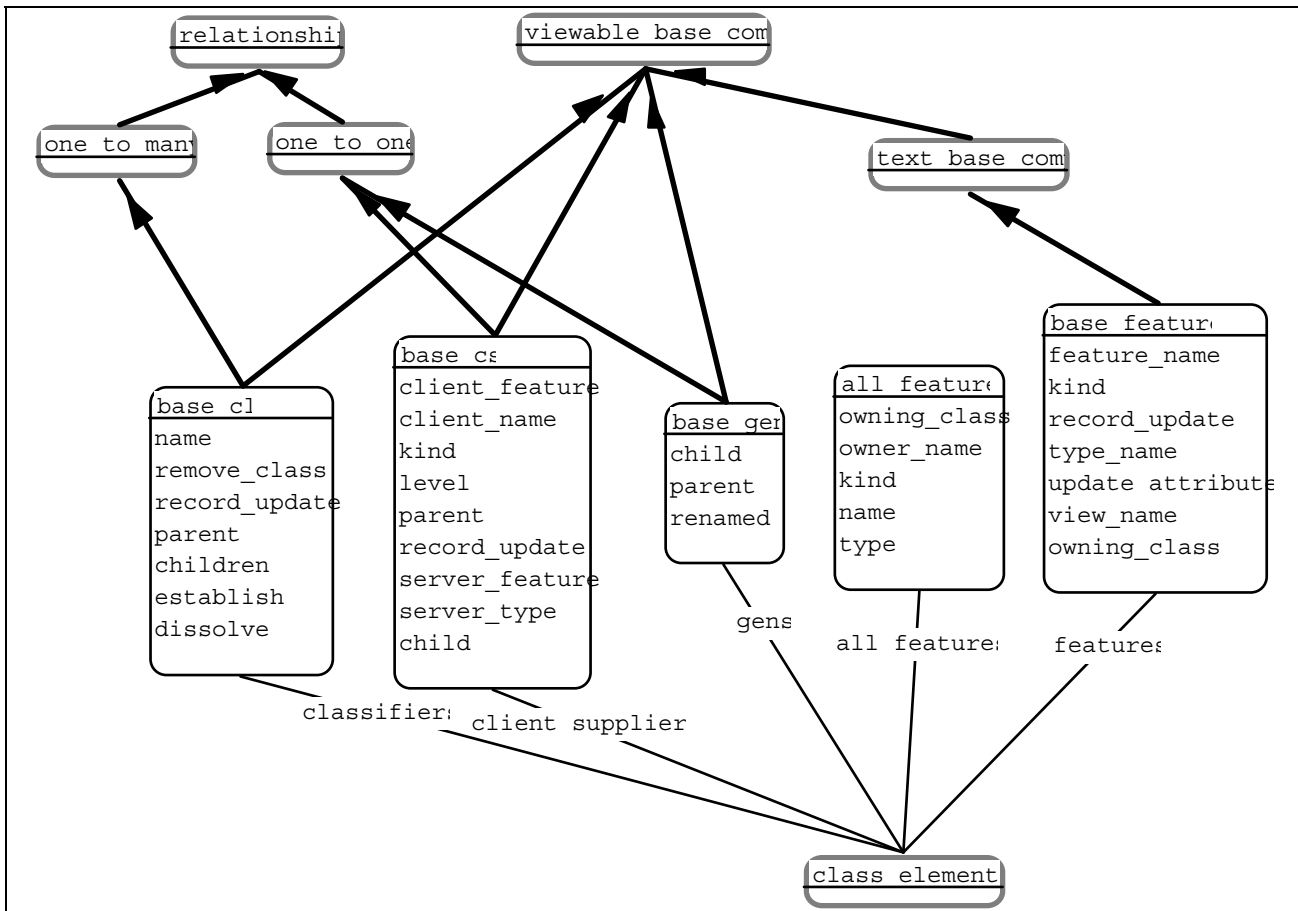
fig. 8.4.   Class component structures and methods.

Generalisations

MVSL generalisations are base relationships and are modelled as relationship components by the IspelM architecture. `base_gen` is a specialisation of both `one_to_one` (i.e. represents a one-to-one relationship component) and `viewable_base_comp` (as generalisations are viewable but do not have text forms). The `parent` and `child` attributes inherited from `one_to_one` are over-ridden to be of type base_class (i.e. `base_gen` relates one class (the parent of the generalisation) to another (the child, or sub-class)). This equates to the MVSL `parent` and `child` relationship component declarations for base generalisation

relationships. Generalisations pass updates on themselves to their child (owning) class and don't currently store them against themselves.

Client-suppliers

Client-suppliers are relationship components relating classes by abstract or inherited aggregation, feature calls, or local argument references. A client-supplier may be abstract (design-level), inherited (defined by an ancestor of its owning class) or code-level (i.e. its owning class is a direct client of its supplier class). Client-suppliers define attributes to represent their parent (owning) base class, supplier (child) base class, a level (design, code, or inherited), and a kind (aggregate (i.e. attribute), feature call, or local reference). Aggregates have a client feature name, locals a client feature name/variable name, and calls have client and supplier feature names. All have a supplier type string used to determine the supplier base class (which may include parameterised classes, such as `list(ClassName)`). Fig. 8.5. shows examples different kinds of client-supplier relationship information.
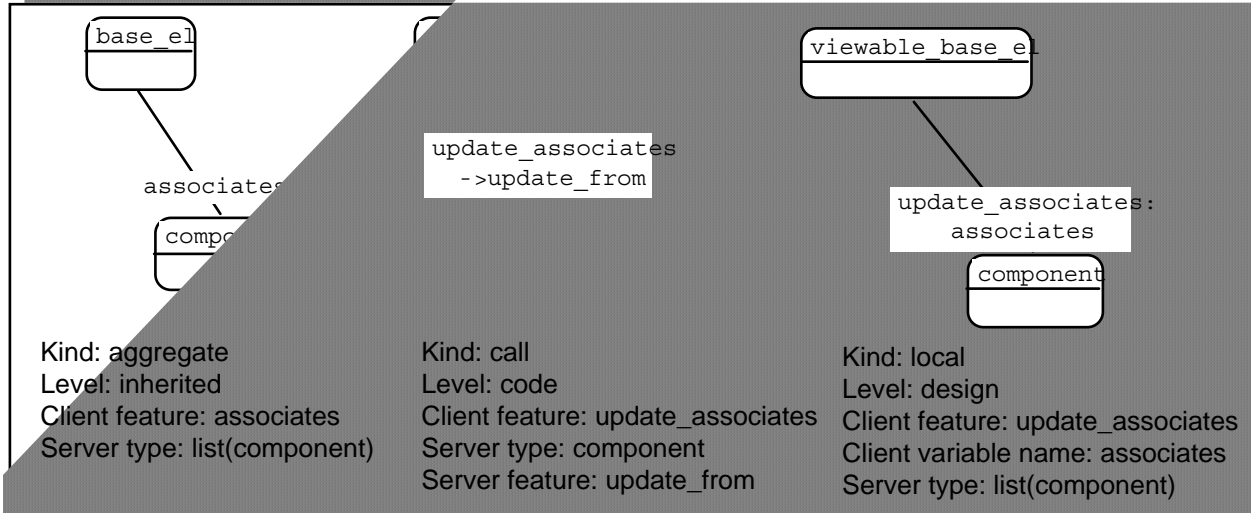
fig. 8.5.   Different kinds of client-supplier relationships and their information.

MVSL client-supplier base relationship components are modelled by the `base_cs` class which, like `base_gen`, is a specialisation of both `one_to_one` and `viewable_base_comp` (for the same reasons `base_gen` is). Client-suppliers pass their update records to their owning class and do not store the records themselves.

Classifiers

MVSL defines classifiers as base relationships and IspelM models these by `base_cl`, a specialisation of `one_to_many` and `viewable_base_comp` from the MViews architecture. One classifier component thus relates its parent (base class) to zero or more children (classifier base classes). `base_cl` provides methods to add and remove classes from the classification

relationships they represent using `establish` and `dissolve` inherited from `one_to_many`. Classifiers pass all updates to their owning class by redefining `record_update`.

8.1.5. Programs

The MVSL specification for IspelM defines a base view called `program` to group base program graph information. The IspelM architecture uses a `program` class which is a specialisation of `base_view`. `program` has a `name` string attribute and one-to-many relationships to the base clusters and base classes it groups (for look-up). `program` defines methods which provide support for adding, removing, locating, and renaming clusters and classes and supports global look-up tables for these components. Base program components can be located given their `unique_id` (using `find_component_id`) or by using a component-specific look-up operation (such as `find_class`). Fig. 8.6. illustrates the `program` class structure and methods for IspelM.


```
program
add_class
add_cluster
compile_program
error_classes
find_class
find_component_id
get_classes
name
remove_class
remove_cluster
rename_class
rename_cluster
```

```
        components          classes
        base cluste         base class
```
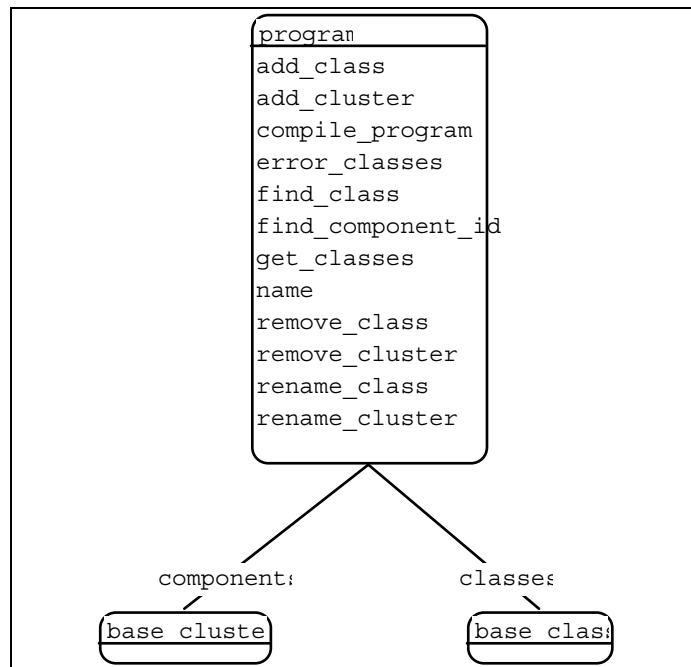
fig. 8.6.   IspelM program structure and methods.

8.1.6. Subset and Display Views and Components

MVSL defines subset views, subset elements, and subset relationships. MVisual defines renderings and interaction mechanisms for the display views and components of these MVSL subset components. The MViews architecture defines display views and components to be specialisations of subset views and components. We can thus define these IspelM subset and display view as specialisations of MViews display view classes. Similarly, we define subset and display view components as specialisations of MViews display component classes.

Class Diagram View

The MVSL specification for IspelM defines a class diagram view as a subset view. It also defines subset components including class icons (subset elements) and generalisation, classifier, and client-supplier glue (subset relationships). MVisual supplies a rendering and interaction specification for class diagram views and for these class diagram view components. The IspelM architecture defines `class_diagram_view` as a specialisation of `graphic_disp_view` and `class_icon` as a specialisation of `graphic_icon`. `gen_glue`, `cs_glue` and `cl_glue` are all specialisations of `graphic_glue`. Fig. 8.7. shows the structure and methods for these classes.

`class_icon` defines a relationship (`feature_names`) and methods to support feature name storage manipulation. It also defines `class_name` and `kind` attributes which mirror those of `base_class`. `map_component` either finds a base class to map a class icon to or creates a new base class from the class icon attribute and relationship information.

`cs_glue` mirrors the attributes of base features and client-supplier relationships. `cs_glue` is used to render features of base classes which equate to code-level client-supplier aggregates. `cs_glue`, `cl_glue` and `gen_glue` define `map_component` methods which find or create appropriate base components to map to. `map_component` is used rather than defining `establish_rel` for class icons to do this mapping as `cl_glue` and `cs_glue` require extra relationship component information to be initialised to identify the appropriate base component to map to.
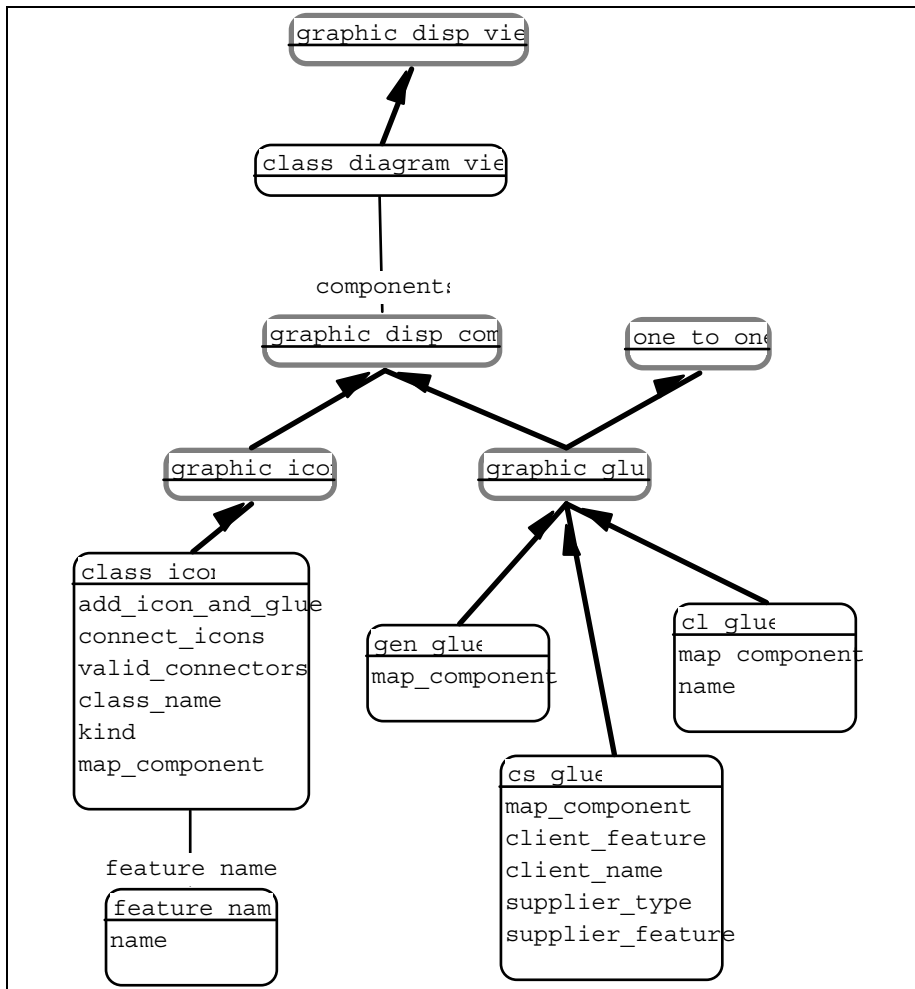
fig. 8.7.   Class diagram view and element structures and operations.

Class Text View

The IspelM architecture defines `class_text_view` to represent class text view subset and display views defined by MVSL and MVisual respectively. `class_text_view` is a specialisation of `textual_disp_view` from the MViews architecture. Class and feature text view components are defined as `class_text` and `feature_text` classes, both specialisations of `textual_disp_comp`. `class_text` defines a class name attribute used to map class textual display components to a base class. `feature_text` defines class name and feature name attributes used to map feature textual display components to a base feature.

8.1.7.   Subset/base relationships

The IspelM architecture defines subset/base relationship classes for each kind of subset component class. These subset/base relationships translate update records generated by subset components into base component operations and vice-versa. All subset/base relationship classes are specialisations of `subset_rel` from the MViews architecture. All IspelM subset/base relationships except client-supplier glue subset/base relationships use

the default action of `subset_rel` when translating subset component attribute updates into base component attribute updates.

Client-supplier subset/base relationships are somewhat more complex in that they allow client-supplier glue to map to either a base client-supplier relationship or base feature component (this is because a code-level, aggregate client-supplier is the same as an attribute (feature) for a class). The client-supplier subset/base relationship defines base_cs and base_feature attributes which it maintains and uses to determine whether it is connected to a base feature or base client-supplier. Update records are translated into appropriate base component and display component operations using these attributes to determine the kind of base component a display component is modelling.

8.2.   IspelM Implementation

IspelM Components	Lines
base_class	16
data	159
components	432
compilation	511
files	525
views	431
base_feature	446
base_gen	212
base_cl	216
base_cs	209
base_cluster	215
class_icon (incl. subset_class)	864
gen_glue (incl. subset_gen)	106
cl_glue (incl. subset_cl)	214
cs_glue (incl. subset_cs)	504
class_text (incl. subset_class)	205
feature_text (incl. subset_feature)	163
class_diagram_view	160
class_text_view	488
program	1011
dialogues	785
application	119
misc. (menus, initialisation, etc.)	124
Total:	8115

table 8.1. Complexity of the IspelM Snart implementation.

8.2.1. Snart Implementation of IspelM

The IspelM architecture described in the previous section illustrates how an MVSL/MVisual specification for IspelM can be translated into a design for implementing IspelM using the MViews architecture of Chapter 6. To implement IspelM the Snart framework of Chapter 7 can be reused. This produces a framework of Snart classes with IspelM architecture classes implemented as Snart classes specialised from MViews framework classes. This framework is itself reusable to produce language-specific software development environments, such as SPE. Table 8.1. illustrates the complexity of the Snart implementation of IspelM by showing a breakdown of code for each Snart class implemented for IspelM.

8.2.2. Base Classes

Snart Base Class

`base_class` defined by the IspelM architecture is implemented as a Snart class `base_class`. `base_class` is the most complex of IspelM's classes and is implemented as five classes implementing different parts of a class's data and behaviour and a sixth class `base_class` which inherits from all of these classes. Originally we implemented `base_class` as one Snart class inheriting from `text_base_comp` but it became so large that modification and recompilation was very time-consuming[25]. Fig. 8.8. shows the extra structure and methods defined by the Snart implementation of `base_class`.

---

[25]This problem suggested a "multiple class view" system may be useful for Snart. This might provide multiple class interfaces for different requirements (typically different class responsibilities) and allow definition of a class over several LPA program windows.
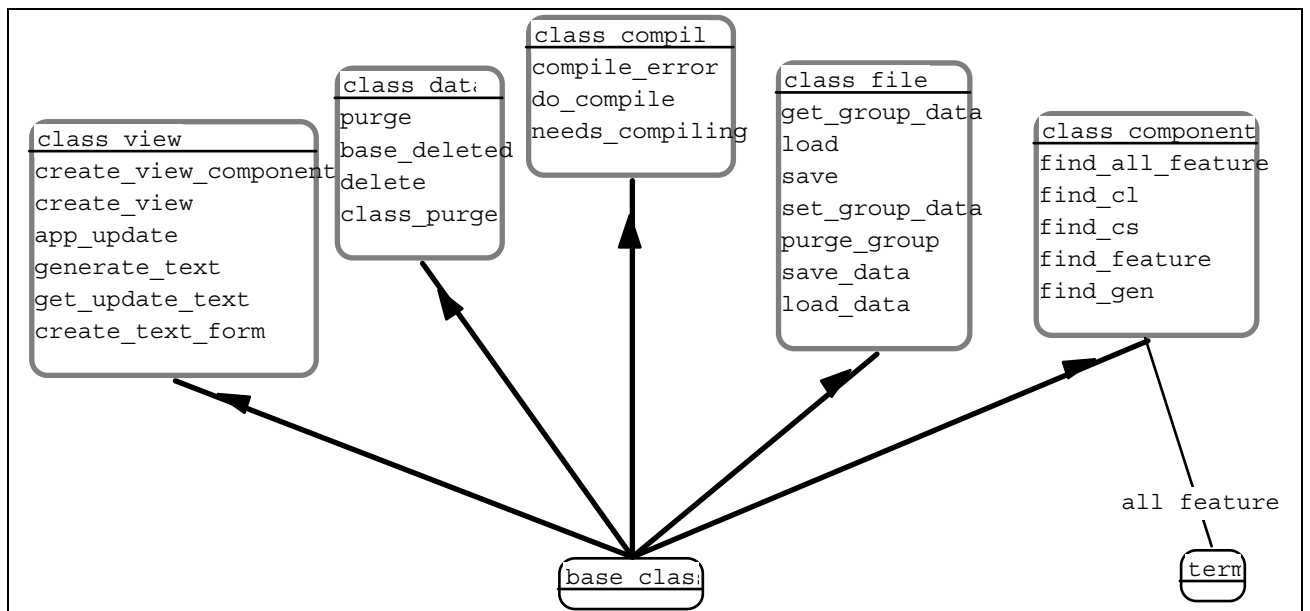
fig. 8.8.   Extra structure and methods for base classes.

Class Data

`class_data` implements the IspelM architecture base class's `class_name` and `kind` attributes as Snart class attributes. `class_data` also implements additional methods for handling class purging, deletion and base view deletion. When a base class is deleted or purged (removed from memory but not from persistent storage) its components (features, generalisations, and so on) must also be deleted. Rather than use `record_update` to pass a `component_deleted(Class)` update record to each component and let it interpret the update, `class_data` calls the `purge` or `delete` methods of each component directly. This is much faster than `record_update` (as a class may have many component objects) but achieves the same result[26]. A special `class_purge` operation is used to keep only the minimum amount of class information in memory.

Class Components

`class_components` stores each kind of class relationship as a list attribute of Snart object references to class component objects (`base_feature`, `base_gen`, etc.) for efficiency. `base_gen`, `base_cs` and `base_cl` relationship component classes are implemented as `viewable_base_comp` specialisations. `base_feature` is implemented as a `text_base_comp` specialisation. Look-up of base features a class owns is via `unique_id` or `feature_name` and

---

[26]`record_update` still generates a `component_deleted(Class)` update record but the base class's components are already deleted by this stage. The Snart framework's support of this application-specific propagation of change, together with the more general `record_update/update_from` form, allows programmers to make a trade-off between abstraction and efficiency as they require.

is implemented as a sequential search of the `features` component object list. This could be implemented more efficiently using a binary search or hashtable but use of IspelM (as SPE) has indicated this simple approach is sufficient for most applications. Base generalisation location is via its `parent.class_name` value and classifier look-up by `name`. Client-suppliers are located by either `unique_id` or `kind`, `level`, `supplier_feature`, `supplier_name`, `client_type`, and `client_feature` values. Only the appropriate values are used in the look-up each kind of client-supplier relationship.

The entire interface for a class is stored as a list of terms of the form `Name(OwningClass, OwnerName, Kind, Type)` in `all_features`. As with update records, using a Snart object to store each inherited feature information is unnecessary (and when used proved very inefficient) as all processing of this data is performed by `base_class`. Look-up of a feature is done by `feature_name` and then `OwnerName` used to find the appropriate feature object reference with `find_feature` for `OwningClass`. When a base class is recompiled, `all_features` is regenerated if the class or one of its ancestor's interfaces have been changed.

Class Compilation

Computation of `all_features` is done in a similar manner to the Snart compiler's determination of a class interface (see Appendix A). Any base classes or components of base classes marked "removed" are sent `delete` messages during this process (thus IspelM uses class compilation to garbage-collect any removed base components). If the interface for a class has changed, a language-specific compiler must be employed to regenerate the dispatch table for the class (and possibly recompile any specialisations and clients of this class). This compilation process is similar for most object-oriented languages but if necessary sub-classing of `class_compile` can over-ride `do_compile`.

Class Views

Class view management extends `text_base_comp` to include creation of class subset views and subset components using `create_view` and `create_view_component`. These methods use the base view `create_component` method to create Snart objects. These objects represent subset views and components for the appropriate kind of subset view a base class owns. Text form creation and validation methods are defined and a declarative `get_update_text` method unparses base class updates records into a human-readable form. A declarative `app_update` method converts update records generated by MViews classes into savable update record terms which don't use Snart object ids (which change when components are reloaded from persistent storage).

Class Files

Base class file management manages saving, loading and purging of class data. When a class is saved it uses group management functions to determine whether various related class components or attributes need saving. Class groups are saved to resources by MViews framework save and load methods. `class_files` implements declarative `get_group_data` and `set_group_data` methods to translate between in-core and persistent group data terms. For example, when a class's features attribute is required and the class's features are not in-core, `get_attribute` calls `load_groups([features])`. `load_groups` loads the feature data (as resource ids) from a class term data file and gives this data to `set_group_data`. `set_group_data` then creates a new `base_feature` object for each resource id and initialises it with the resource data (by calling `load_data` for the `base_feature`).

8.2.3. Class Components

`base_feature` is implemented as a specialisation of `text_base_comp` from the Snart framework for MViews. `base_feature` attributes are implemented as Snart attributes and `record_update` sends base feature update records to a feature's owning class. Base features are saved to their owning class's term data file as a term and reloaded by `save` and `load` methods implemented by `base_feature`.

`base_gen`, `base_cs` and `base_cl` are implemented as Snart classes which inherit from `viewable_base_comp` and `one_to_one` (`one_to_many` for `base_cl`) using multiple inheritance. These relationship component classes use Snart object ids to refer to the base classes they relate. Their attributes are implemented as Snart attributes and they are saved and reload to and from their owning class term data files as single Prolog terms. Restoration of their Snart object references on reloading is done by using the base view look-up tables and unique base class ids.

8.2.4. Programs

`program` is implemented as a Snart class which inherits from `base_view`. `program` uses hashtables for locating classes (for efficiency) and these look-up tables are regenerated as part of the reloading of clusters and classes when a program is re-opened. Compilation and parsing support for IspelM is implemented by calling `parse_view` for updated textual views and `do_compile` for clusters (and thus classes). When a program is saved or compiled IspelM calls a `class_purge` method for base classes to ensure only necessary information is held in-core.

`program` implements a "grass-catcher" similar to that provided by the Trellis/Owl environment (O'Brien et al 87) for locating base classes with semantic or compile errors. Compile-time and semantic errors are of the form `semantic_error(Kind,Data)` and

`compile_error(Kind,Data)` and are generated by the `base_class do_compile` method and semantic calculation methods driven by `update_attribute` for `base_class`.

8.2.5. Subset and Display Views and Components

Class Diagram Views

`class_diagram_view` is implemented as a Snart class which inherits from `graphic_disp_view`. `class_diagram_view` defines tools and methods which implement these tool. Graphical display view tools are implemented by defining LPA MacProlog GDL (Graphic Description Language) pictures[27] to represent the tool and implementing methods for tool functionality. The `graphic_disp_view` class provides generic `add_icon` and `add_glue` methods as well as "manipulators" (of a similar nature to those of (Linton et al 88)) for implementing line connection, icon dragging and double-clicking, marqui selection, cut/copy/paste functions and component hiding and removal. Fig. 8.9. shows the extra class diagram view and component structure and methods used by the Snart implementation of IspelM.

`class_icon` is implemented as a Snart class inheriting from `graphic_icon`. `get_picture` for `class_icon` returns a GDL description for a class icon. This picture is composed by using data held in a `class_icon` object to produce GDL picture for the class icon. `draw_component` inherited from `graphic_icon` uses LPA MacProlog predicates to draw and manipulate this picture.

Class icons implement their feature names attribute as a list attribute of Prolog atom values (i.e. strings). Originally, this was implemented by using sub-icons for each feature name. This approach, however, proved to be slow in response time (due to many Snart objects needing to be created) and more complex then necessary. `class_icon` implements methods to add, remove and change feature names. Double-clicking on a class icon performs the actions described in Chapter 4. Click-points are defined for browsing a class's views, selecting a class's default text view, selecting feature views, and selecting from a class's features (either those owned by the class or from its full interface). Connection processing creates new glue between class icons to represent generalisations, client-suppliers (including features) and classifiers.

---

[27]See Appendix A for examples of such GDL pictures.
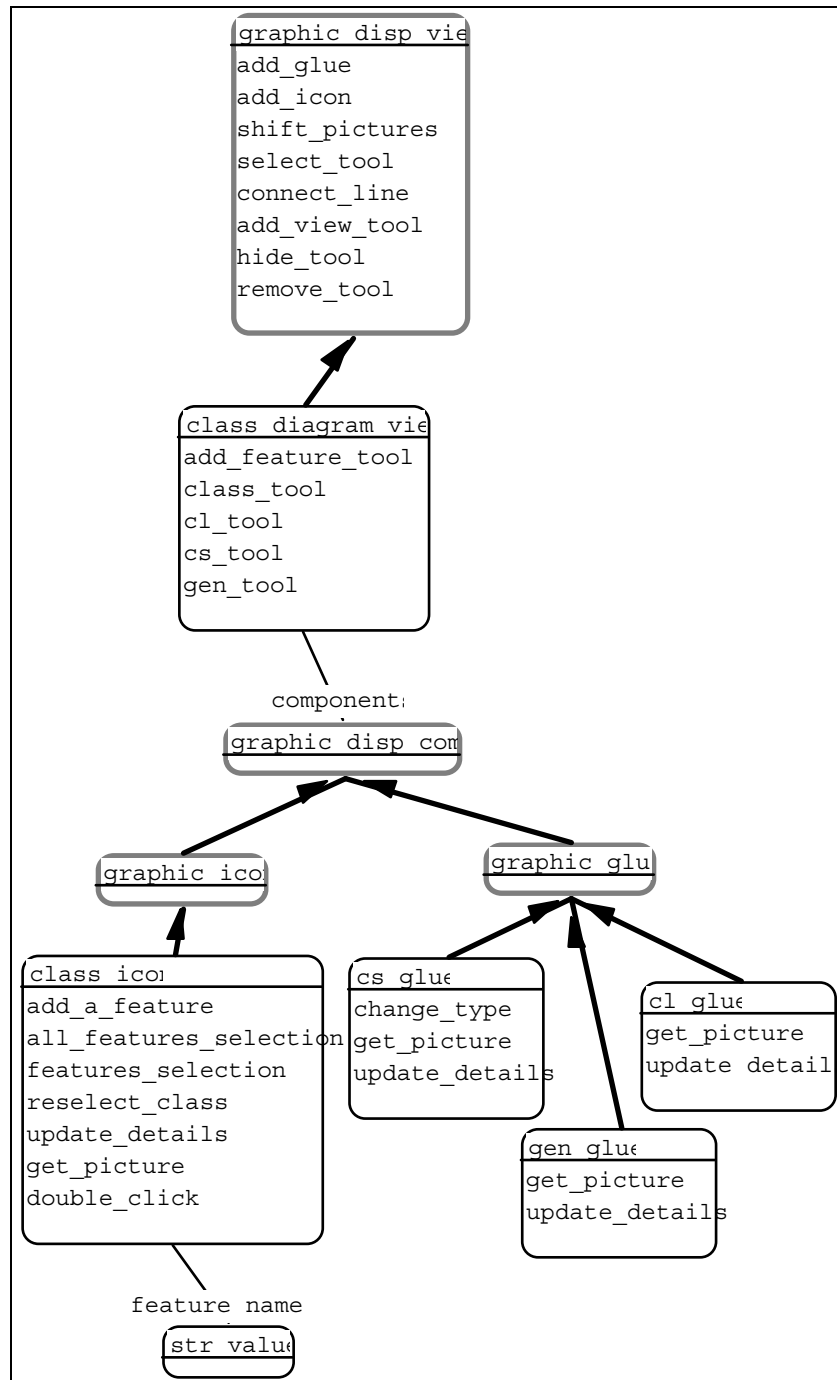
fig. 8.9.   Extra class diagram view and component structure and methods.

Client-supplier glue (implemented by a `cs_glue` class which inherits from `graphic_glue`) provides an `update_details` method which allows the `supplier_type` value for a client-supplier relationship to be modified. If this value is changed the class icon acting as the supplier must be remapped to a different base class. This is achieved by having the client-supplier subset/base relationship call `change_type` for its subset component(s) and the subset client-supplier glue calls `reselect_class` for affected class icon subset components. `reselect_class` is implemented by unmapping a class icon from its old class and remapping it to a new base class. All class icon feature names and connector glue is re-validated by this process by checking feature names against the new base class's

`all_features` list and re-mapping glue to base components. Any inconsistent feature name values and glue components are deleted or left unmapped (and re-rendered to indicate this) respectively. Client-supplier, generalisation and classifier glue classes implement `get_picture` methods which return GDL pictures to act as renderings of these graphical display relationship components.

Class Text Views

`class_text_view` is implemented as a Snart class which inherits from `textual_disp_view`. Class text views implement language-specific parsers which produce Prolog terms by parsing text associated with class and method text forms in the view. These terms are given to a `process_term` method which computes changes in the base class/method information from the parsed data. Changes are determined by first generating terms equating to the current class/method state (class interface information for classes and method name and interface information for methods). These terms are compared with the parsed data terms and changes computed. For example, if a class feature does not appear in the parsed term list but does in the current class `features` relationship then the feature has been removed and its corresponding base feature must be removed. These changes are applied directly to affected base components by calling methods for their objects.

Class text views allow new text forms to be expanded into the view and base data to be expanded into a text form (such as class feature names and types). These facilities are implemented by providing selection dialogues for the information to expand and then either unparsing the information (for data expansion) or adding text forms to the view.

8.2.6. Subset/base relationships

Class, feature, generalisation and classifier subset/base relationship classes implement a base component to subset component attribute correspondence method `base_to_subset` as lists of terms of the form `[BaseAttribute-SubsetAttribute,...]`.

Client-supplier subset/base relationship class `cs_subset_rel` redefines `process_update_from_base` and `process_update_from_display` methods inherited from `subset_rel`. `cs_subset_rel` determines the kind of base component it is connected to (feature or client-supplier) before translating updates. This is done in a declarative manner using update records sent to the subset/base relationship and values of base_cs and base_feature attributes (updated when the relationship is established by its subset component). For example, an `update_attribute(Glue, client_feature, NewName)` update record from a client-supplier glue object must be converted into an `update_attribute(Feature, feature_name, NewName)` operation on a base feature object (and vice-versa).

8.2.7. Update Records

IspelM operations make use of the fundamental operations supplied by the MViews framework and these MViews operations generate the set of update records described in Section 5.3.3. IspelM components generate, propagate and respond to these fundamental update records using behaviour they inherit from the MViews framework classes. IspelM components store these update records in an application-specific form, however, so IspelM components can be made persistent without using object IDs which may change from one invocation of the environment to the next. IspelM also uses these stored update records to form the update history for components and for unparsing into textual display views. Table 8.2. shows how MViews' fundamental update records are stored by IspelM and the form these IspelM update records are unparsed into a textual display view or update history dialogue.

MViews Update Record	IspelM Update Record	Textual view/update history form	Description
update_attribute(Class, class_name,Old,New)	rename_class(Old,New)	% rename Old to New	Rename Class from Old to New
update_attribute(Class, kind,Old,New)	change_kind(ClassName, Old,New)	% change class kind to New	Change Class kind from abstract to normal or vice-versa
update_attribute(Feature, feature_name,Old,New)	rename_feature(Old,New)	% rename feature Old to New	Rename Feature from Old to New
update_attribute(Feature, type_name,Old,New)	change_type(FeatureName,Old,New)	% change type of FeatureName to New	Change Feature type from Old to New
update_attribute(Feature, kind,Old,New)	change_kind(FeatureName,Old,New)	% change kind of FeatureName to New	Change Feature kind to attribute, method, deferred, or inherited
update_attribute(CS, Attribute,Old,New)	change_cs(CSName, Attribute,Old,New)	% change CSName Attribute to New	Change a client-supplier relationship Attribute to New
update_attribute(Cl, name,Old,New)	rename_classifier(Cl, Old,New)	% rename classifier Old to New	Rename a classifier
establish(classes, Cluster,Class)	add_class(ClusterName, ClassName)	% add class ClassName	Adda class ClassName to a cluster ClusterName
establish(features, Class, Feature)	add_feature(ClassName , FeatureName)	% add feature FeatureName	Add feature FeatureNamem to class ClassName
establish(gens, Class, Gen)	add_gen(ClassName, ParentName)	% add generalisation to ParentName	Add generalisation from class ClassName to parent class ParentName
establish(css,Class, CS)	add_cs(Classname, CSName)	% add client-server CSName	Add client-server CSName to class ClassName
establish(cls,Class, Cl)	add_cl(Classname, ClName)	% add classifier ClName	Add classifier ClName to class ClassName
establish(rename,Gen, Rename) dissolve(rename,Gen, Rename)	add_rename(ParentName, Rename) remove_rename(ParentName,Rename)	% add rename from ParentName Rename % remove rename from ParentName Rename	Add/remove rename of a ParentName class feature

establish(classifier, Cl,Class) dissolve(classifier, Cl,Class)	add_classify(ClName, ClassName) remove_rename(ClName, ClassName)	% add classify to ClassName using ClName % remove classify to ClassName using ClName	Add/remove classification to class ClassName using classifier ClName

remove(Class) unremove(Class)	remove_class(ClassName) unremove_class(ClassName)	% remove class ClassName % unremove class ClassName	Mark/unmark a base class as "removed"
remove(Feature) unremove(Feature)	remove_feature(FeatureName) unremove_feature(FeatureName)	% remove feature FeatureName % unremove feature FeatureName	Mark/unmark a base feature as "removed"
remove(Gen) unremove(Gen)	remove_gen(ParentName) unremove_gen(ParentName)	% remove generalisation to ParentName % unremove generalisation to ParentName	Mark/unmark a generalisation to parent class ParentName as "removed"
remove(CS) unremove(CS)	remove_cs(CSName) unremove_cs(CSName)	% remove client-supplier CSName % unremove client-supplier CSName	Mark/unmark a base client-supplier relationship as "removed"
remove(Cl) unremove(Cl)	remove_cs(ClName) unremove_cs(ClName)	% remove classifier ClName % unremove classifier ClName	Mark/unmark a base classifier as "removed"

table 8.2. IspelM update records.

8.2.8. User Interaction

Menus

The Snart implementation of IspelM uses LPA Prolog predicates to define extra menus for textual and graphical display views. These call Prolog predicates which in turn call display view `process_menu_item` methods.

Dialogues

The IspelM implementation uses LPA Prolog to implement dialogues which support object-oriented program browsing and manipulation. Examples of the feature selection and feature definition dialogues for class icons are shown in fig. 8.10. (see Chapter 4 for the purpose of these dialogues).
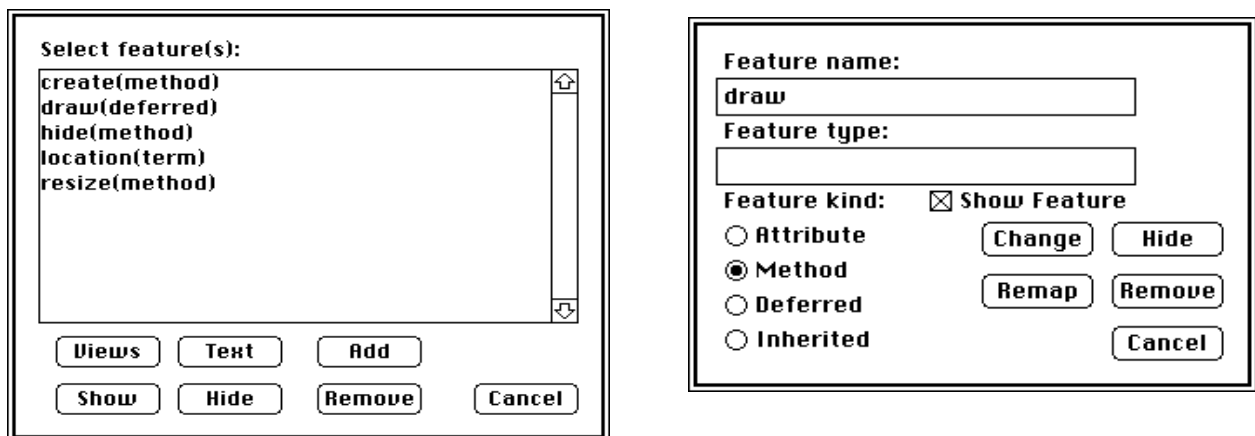
fig. 8.10. The feature selection and feature definition dialogues for class icons.

Dialogue opening methods and predicates supply information to LPA dialogue predicates and receive information back in Prolog variables. A method or predicate which opens a dialogue can call component object methods to access component attribute and relationship values and update component attributes.

8.2.9. Program Persistency and Execution

The `program` class implements `process_menu_item` methods to support save and load program operations. `program` saves its components (clusters), clusters save their classes and classes save their components. All IspelM program component data is stored in term data files. Subset and display view data is assumed to be saved in term data files associated with the focus component for the views. The program component, base clusters, and base class instances each have a term data file. Class components are saved to the term data file associated with their owning base class.

The IspelM framework assumes an interface to a language-specific compiler is implemented. This allows base classes to either be generated from class information supplied by IspelM or to compile textual code views to regenerate their executable program data. Executing a program is assumed to be via a Prolog predicate call and `program` implements methods to execute a program and delete objects created by a program.

8.3. The Snart Programming Environment

To provide a software development environment for Snart programming, the IspelM framework must be extended by specialisation to produce SPE. IspelM does not directly support any form of language parsing, dynamic language semantics, or interface to an object-oriented language compiler or run-time system. SPE extends IspelM to produce an environment suitable for Snart programming by sub-classing from the IspelM framework. Fig. 8.14. shows the extra classes defined for SPE.
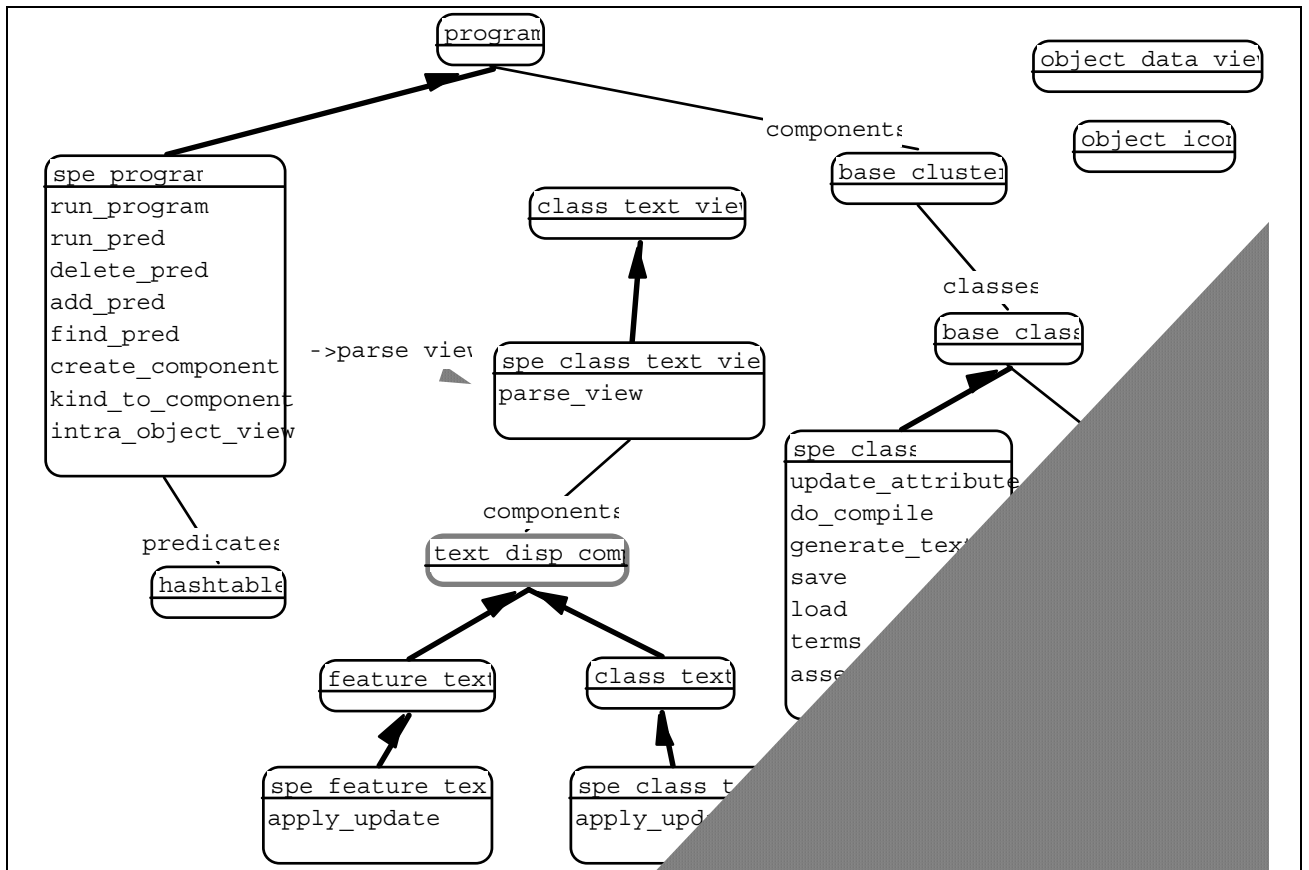
fig. 8.14. Extra classes defined to implement SPE.

8.3.1. Parsing and Unparsing of Snart Programs

Parsing class definitions and method predicates generates lists of terms representing the parsed information. A class definition produces generalisations (with renamed features) and features (feature name/type pairs). These lists are compared against the base class information by IspelM and base updates performed as necessary. `spe_class_text_view` specialises `class_text_view` to provide this parsing and updating support for SPE. Update application is done by incremental token parsing and substitution based on Snart syntax and `spe_class_text` and `spe_feature_text` specialise `class_text` and `feature_text` to provide Snart-specific update application.

When generating text forms for class text views SPE must unparse base class and method data into Snart class and method predicate syntax. A class and method template is used to layout the new text form and class and method interface data unparsed into the appropriate place in this template. `spe_class` and `spe_feature` redefine `generate_text` to support Snart-specific text form generation for class definitions and methods.

8.3.2. Validation, Compilation and Saving of Snart Programs

Snart classes can not have the same name as any Prolog predicate (as Snart generates a term using the class name to store compiled class information). SPE ensures classes are not

named as or renamed to existing predicate values adding a predicate look-up table to `spe_program`, a specialisation of `program`. `spe_class` and `spe_feature` also maintain lists of terms defined by their text forms.

`spe_program` defines new `kind_to_component` and `create_component` methods so SPE-level objects are created by IspelM and SPE classes. For example, rather than `map_component` for a `class_icon` creating a `base_class` if one does not exist, it creates an `spe_base_class` by calling `spe_program`'s `create_component` method.

SPE can assert Prolog terms (for example, Snart method implementations) and call the existing Snart compiler to regenerate its data structures based on these new terms. Alternatively, the Snart compiler can re-compile textual display view text windows and thus generate updated class information. The first approach involves more work for an implementer of environments as compilers may not directly allow such incremental updates of a program data. It is more efficient than the second, however, which requires both the environment and the language compiler to parse a textual view and update their data structures and compiled code independently. SPE currently uses the first approach and calls Snart compiler predicates to regenerate Snart class dispatch tables.

`save` for `spe_class` checks if predicates or Snart class information needs to be saved for a Snart program. `load` reloads compiled Prolog terms from a term data file so Snart programs developed in SPE can be executed from within the environment.

8.3.3. Running and Debugging Snart Programs

`spe_program` provides program execution facilities by implementing a "run predicate" to execute a Snart program and a "delete predicate" to clean up objects created by a Snart program. As SPE uses Snart objects to store its data, the object space of a running program and SPE must be distinguished by the Snart run-time system. The run predicate creates and initialises an object space for the program and deletion predicate deletes all Snart objects associated with the program (as well as any windows created by the program).

SPE programs are debugged visually and textually. An object data view allows Snart objects to be displayed and navigated between using graphic display views (showing the state of a single Snart object) and the Prolog debugger is used to trace method and predicate execution. Objects can be displayed by selecting their object id from the debugger window and using a menu option, entering an object id, or double-clicking on an object reference in an object data view. Object data views obtain object attribute values via the Snart run-time system which provides a dynamic access function of the form `default_value(ObjectID, AttributeValue, DefaultValue, Value)`. `object_data_view` is a specialisation of `graphic_disp_view` while `object_icon` is a specialisation of `graphic_icon`.

8.4. Discussion and Future Research

8.4.1. IspelM Model

Program Representation

IspelM reuses the MViews architecture to model the structure and some high-level static semantics for object-oriented programs. The MVSL definition of IspelM translates into component and relationship component classes specialised from MViews architecture classes. These IspelM classes define extra attributes, relationships and methods for object-oriented program structures and structure manipulation. For program structure the IspelM model works well and it can represent most important, high-level aspects of class-based object-oriented languages as specialisations of MViews component classes. Documentation and method implementation detail can be represented as text forms, as can additional class interface information.

This model can be extended quite naturally to incorporate language-specific features such as class contracts for Eiffel (Meyer 92), information hiding for Kea (Hosking et al 90), C++ (Stroustrup 86) and Eiffel, classifiers (Kea), and attribute typing for all strongly-typed object-oriented languages. These additional features would necessitate defining new component classes as specialisations from MViews classes with appropriate attributes, relationships and methods. Some features which would be required for most other class-based languages and are not currently modelled include generic classes[28], method arguments, and typing (and type checking) for method arguments and feature calls.

IspelM can model languages such as Smalltalk (Goldberg and Robson 84) and CLOS (Keene 89) which use class-as-object representations by creating class objects based on its internal representations (similar to how SPE creates Snart class definition predicates for the Snart compiler to use). It is not clear, however, how suitable IspelM's program representation is for non-class-based object-oriented languages, such as SELF (Ungar et al 92), which use prototypes and traits objects to model object behaviour.

Some semantic calculation (such as inherited class interfaces) and constraints (such as unique feature names per class) are captured well by IspelM. Other values, such as the dynamic semantics of object-oriented programs, are more difficult to specify using MViews' object and attribute dependency mechanisms. Comparable approaches, such as Kaiser's action equations (Kaiser 85) or Hudson's (Hudson 91) and Reps' (Reps and

---

[28]IspelM does support simplistic, single parameter generics, such as `list(ListType)`, for collection classes. This should be extended so any parameterised class can be modelled including multiple parameter type values.

Teitelbaum 87) attribute recalculation schemes, would offer more abstract and declarative ways of specifying such language behaviour. The Mjølner environments (Magnusson et al 90) use a structure editor for representing and manipulating program structure (Minör 90) and attribute grammars which are sent structure editing operations to model static and dynamic semantics. A similar approach with MViews update records being used to drive an attribute recalculation scheme is discussed in Chapter 10.

Views and View Components

The MVSL and MVisual specifications of IspelM defined basic program views for visualising and constructing object-oriented programs. These include views for class structure and basic control flow (class diagrams), adding class and method detail, documentation, and possibly design information (textual views) and provide very basic object data visualisation and navigation (object data views). The IspelM architecture defines classes specialised from MViews architecture classes to model these views and view components.

As discussed in Chapter 4, SPE (and IspelM) provide only a limited number of program visualisation and construction views. Additional views might include inter-feature relationships (including call sequencing and argument data), class contract views, and improved program visualisation. These extra views could be defined using MVSL and MVisual and translated into IspelM architecture classes in the same manner as class diagram and class text views.

Abstraction

Some aspects of IspelM architecture classes could be better represented as part of (or specialisations of) MViews classes. Program navigation is partly abstracted out as MViews class methods which support click-points and menu dialogues for view navigation. Expansion of class details (such as feature names or client-supplier relationships) is mostly implemented by IspelM classes whereas ideally MViews classes should provide expansion and menu selection methods reusable by IspelM and other environments.

8.4.2. IspelM Implementation

The IspelM architecture is implemented as a Snart framework of classes which reuse the MViews Snart framework described in Chapter 7. This produces a framework of classes suitable for constructing language-specific object-oriented programming environments, such as SPE. Reusing the MViews Snart framework for IspelM greatly simplified implementation of the environment. MViews' base component classes were easily reused to represent object-oriented program structures. Methods for manipulating these structures were incorporated into IspelM classes and reused methods from MViews classes. Display view classes and components defined by the IspelM architecture were

implemented as specialisations of MViews Snart classes. The MViews classes provided much of the user interface and persistency functionality for IspelM classes.

The main areas of effort in developing IspelM were concerned with appropriate interaction and representation schemes (particularly for graphical class diagram views), program complexity management and navigation, program compilation and constraint semantics, and program persistency management. Graphical views required some effort to determine both the rendering of program components and how best to interact with these graphical forms. Translating dialogue interactions into base component and subset view component method calls also required some work. In particular, while LPA provides good dialogue specification predicates, specifying dialogue layout and behaviour are still left entirely to the implementer of IspelM. An interface builder and constraint system may be useful extensions to MViews to simplify dialogue and graphical view construction (discussed further in Chapter 9).

As MViews does not currently abstract out much complexity management or navigation IspelM must implement these itself. Such techniques are probably useful in other systems (see Chapter 9) and thus extra support at the MViews level should be provided. A problem with this kind of abstraction, and view navigation and creation currently supported by MViews, is tailoring dialogues and interaction to specific environments. For example, specialisations of IspelM currently must redefine dialogues for display view creation to conform to the application's use of display views.

A similar problem arises with IspelM-level dialogues for feature specification and expansion. IspelM treats all class features the same for representation in class icons but distinguishes between "methods", "attributes", "deferred", and "inherited" features in some dialogues. SPE also requires additional constraints to be added to dialogue text fields so valid Prolog atom values are used. Some languages, such as Kea (Hosking et al 90), do not make a distinction between class features, and some modelling techniques, such as MOSES (Henderson-Sellers and Edwards 90), do distinguish between feature name kinds in class icons at the analysis or design levels. More research is required to determine suitable ways of allowing dialogues to be "configured" in sub-classes to support application-specific dialogue tailoring without having to completely re-implement dialogue layout and behaviour.

Language constraints and semantics are not implemented particularly abstractly using MViews' attribute and component dependencies. A more declarative style would be more easily understood and extended than over-riding MViews methods and writing the code in Prolog.

This MViews framework persistency mechanisms provide great flexibility and efficient use of memory (as program data can be incrementally loaded, saved and purged).

Experience with implementing IspelM and SPE using the MViews framework has indicated a need for a more abstract approach to MViews component persistency. The main disadvantages with the current approach include: specialisation classes must implement their own group management methods; classes must implement methods to relink reloaded components; and specialisation classes must synchronise saving, loading and purging of program data. An improvement might use an object-oriented database for storing MViews data with in-core and persistent objects managed automatically by MViews or the database (see Chapter 10 for further discussion of extending MViews persistency management).

8.4.3. The Snart Programming Environment

Most of the effort in specialising IspelM to SPE involved adding language-specific parsing and unparsing, interfaces to the Snart compiler and run-time system, and extra support for saving and reloading Snart executable code. Most IspelM dialogues are suitable for SPE and thus did not require modification for Snart programming[29].

As noted in Chapter 7, a closer relationship between parsing and unparsing (both incrementally and for generating text forms) would simplify tailoring IspelM for different languages. In particular, for languages such as C++ or Eiffel a lot of effort would be needed to write parsers to produce the Prolog structures used by IspelM for updating base information. Additional support at the MViews level would also be needed to support fine-grained text forms (for example, for class contract information).

Adding types to Snart, as suggested in Chapter 3, would allow more compile-time checks and optimisations to be made and provide valuable information for SPE to generate client-supplier relationships (for call-graphs and other view expansion/navigation facilities). This would require intra-term parsing of Snart code to determine appropriate types and strip out the type information (as it is not required for actually executing Snart programs). This would require IspelM to be extended to cope with language types but, as a side-benefit, this would make IspelM more applicable for programming strongly-typed object-oriented languages.

8.5. Summary

An object-oriented architecture for IspelM has been defined using the MVSL and MVisual specifications for IspelM from Chapter 5 and by reusing the object-oriented architecture of

---

[29]As noted previously, for other languages and object-oriented modelling techniques some modification of IspelM's views and dialogs would be required to suit the particular application language and CASE methodology.

Chapter 6. IspelM reuses MViews architecture classes to produce a design for a novel, integrated software development environment for object-oriented languages supporting multiple textual and graphical views with consistency. IspelM allows an object-oriented program to represented as an object dependency graph, be viewed and manipulated in graphical and textual forms, be incrementally saved and loaded, and supports flexible program visualisation, navigation and complexity management.

An implementation of IspelM as a framework of Snart classes is produced by reusing the Snart framework form MViews described in Chapter 7. IspelM program and view component classes are implemented by sub-classing appropriate MViews framework classes. User interaction is provided by display views and components which implement specialised display view, icon, glue and text component classes. These are manipulated by tools for graphical views, typing and parsing text for textual views, and by using dialogues and menus built using LPA predicates. IspelM is specialised to SPE by defining new Snart classes which inherit from IspelM framework classes. These specialised classes implement parsers and unparsers for Snart syntax, an interface to the Snart compiler and run-time system, and compiled Snart code saving and loading support.

Development of IspelM and SPE architectures and implementations indicated that the MViews architecture and frameworks of Chapters 6 and 7 significantly enhance construction of software development environments. Providing a set of reusable classes based on the MViews model of Chapter 5 allows new environments to be defined using MVSL and MVisual and then an implementation designed by reusing the MViews architecture. To implement an environment the Snart framework for MViews is reused. This architecture and framework provide classes which abstract out the data storage, multiple view support, change propagation, and some persistency management aspects of software development environments based on the MViews model.

IspelM is suitable for specialisation to produce an environment for other class-based object-oriented languages and SPE could be extended to support "typed" Snart programs. The MViews architecture and implementation could be extended to better capture program browsing and complexity management and provide enhanced support for dialogue interaction. MViews should also provide more abstract component persistency management for environments. This should support tools for version control and multi-user, shared access to programs.

Chapter 9

Further Applications of MViews

Chapter 8 illustrated use of the MViews architecture and framework for modelling and implementing IspelM and SPE. This chapter demonstrates that the MViews model for integrated software development environments can be reused for several other diverse applications. The concept of multiple textual and graphical views of information with automated consistency management is useful in many different applications. These include, but are not limited to, entity-relationship modelling, dialogue painting, program visualisation, debugging and animation, dataflow diagrams and programs, and more detailed object-oriented analysis and design. Such applications can reuse MViews' novel aspects of flexible information storage (using base view components and relationships), multiple textual and graphical view abstractions, and propagation and documentation of change (using update records) in quite different ways.

This chapter describes several applications developed using MViews. Some have been implemented using the Snart framework from Chapter 7 while others are abstract designs illustrating how a system can be modelled using MViews abstractions. Some systems have been designed and implemented by the author while others are being developed by other researchers using MViews. An entity-relationship modeller provides graphical entity-relationship diagrams and corresponding textual relational database schema for entities and relationships. A dialogue painter provides a graphical dialogue painting view and one or more textual views which define dialogue semantics and constraints. Program visualisation using Snart and MViews is illustrated with a tally graph view of object method calls, sorting algorithm animation, and a visual debugger for SPE. Extended object-oriented analysis and design in SPE includes graphical method calling views and textual class contract views. Dataflow diagrammers support dataflow modelling for analysis and dataflow method implementations for SPE. Other systems include a common building model represented and manipulated using derivatives of SPE and its visual debugger and tool abstraction using MViews component specialisations.

9.1. Entity-Relationship Modelling

9.1.1. Entity-Relationship and Relational Database Modelling

Entity-relationship (ER) modelling (Chen 76) is typically used to model database systems by decomposing data into *entities* and *relationships* between entities. Entities and relationships may have associated *attributes* and relationships can specify a *cardinality* between related entities. An ER model can be successively refined to form the basis for a relational database schema (RDS) (Teorey et al 86), typically composed of *tables* and table *fields*.

ER models can be constructed and queried by graphical techniques (Czejdo et al 90, Teorey et al 86), as can RDSs (Larson, 86). A typical approach, however, is to translate high-level graphical ER (or extended ER) models into low-level textual RDS definitions (Czejdo et al 90). One disadvantage with this approach is that extra information defined by RDSs (such as keys, default values, constraints on table field values, and so on) must be specified externally or added in some way to the ER model.

9.1.2. MViewsER

One solution to integrating ER and RDS specification is to provide graphical ER modelling views and complementary textual RDS views, with consistency management between the two. Fig. 9.1. shows an example of MViewsER, which takes this approach to database model specification.
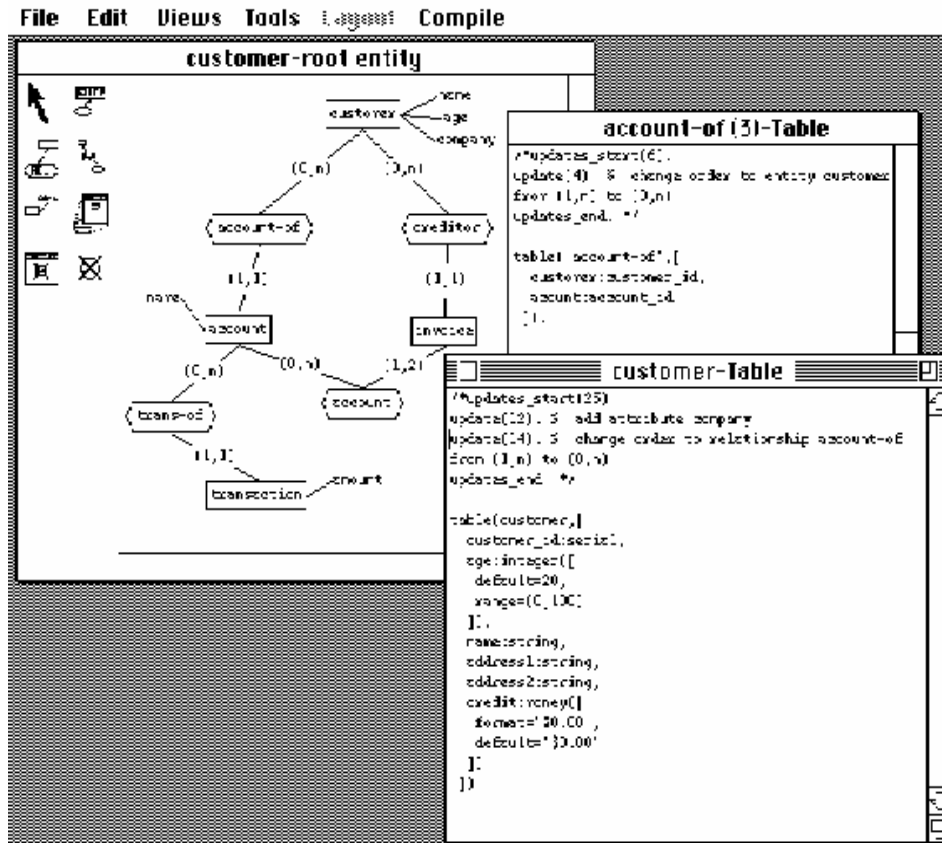
fig. 9.1.   MViewsER graphical ER views and textual RDS views.

MViewsER supports graphical ER diagram views with diagrams constructed using tools, dialogues and menus. Textual RDS views contain a table definition including table fields, field types, and zero or more field values used to specify various attributes for fields. RDS views are parsed to update table information. The graphical ER views provide a high-level specification system with details about RDS requirements ignored. Textual RDS views can be generated from ER data and provide extra information about field types, defaults, ranges and so on.

Consistency management is employed between ER diagrams and RDS tables. Currently MViewsER assumes entity and relationship names and attribute names map to corresponding RDS table names and field names. ER diagram views are updated directly by changes to RDS table views and an update history is kept for entities and relationships (and their corresponding RDS tables). RDS views are updated by unparsing update records (as shown in fig. 9.1.), and some update records can be automatically applied by MViewsER to reflect changes to entities and relationships. Other update records serve as documentation to inform programmers of ER model changes that may or may not impact on the RDS tables.

9.1.3. Specification

MViewsER was initially specified using the MVSL and MVisual notations from Chapter 5. The first task was to determine the base view and subset view components required by MViewsER. An ER model is composed of entities and relationship elements which are related by connection relationships (which hold the cardinality from the entity to the relationship and possibly a name used by the relationship to refer to the entity). Entities and relationships can hold attribute values which have a name, type and list of values. This analysis results in MVSL base component specifications, some of which are shown in fig. 9.2.

Two kinds of subset views are defined by MViewsER. ER diagram views contain entity, relationship and attribute icons and connection and attribute glue. RDS views contain table text forms. Fig. 9.3. illustrates some MVSL definitions for these view components.

```
base element entity
   attributes
     entity_name : string
   relationships
     relationships : connection.parent
     attributes : one-to-many attribute
   ...
end entity

base element relationship
   attributes
     rel_name : string
   relationships
     % note that a relationship may relate > 2 entities i.e. not necessarily binary
     entities : connection.child
     attributes : one-to-many attribute
   ...
end relationship

base relationship connection
   parent entity
   child relationship
   attributes
     order_entity : integer
     order_rel : integer
     name : string
   ...
end connection

base element attribute
   attributes
     attr_name : string
     attr_type : string
     attr_values : one-to-many attr_value
   ...
end attribute
```

<div align="center">fig. 9.2.   Base component MVSL specification for MViewsER.</div>

```
subset element entity_icon
   attributes
     entity_name : like entity.entity_name
   relationships
     base_entity : one-to-one entity
   ...
end entity_icon

subset element attr_icon
   attributes
     attr_name : like attribute.attr_name
   relationships
```

```
      base_attr : one-to-one attribute
   ...
end attr_icon

subset relationship con_glue
   attributes
      order_entity : like connection.order_entity
      order_rel : like connection.order_rel
   relationships
      base_con : one-to-one connection
end attr_glue

subset element table_text
   relationships
      % mutually exclusive
      base_entity : one-to-one entity
      base_rel : one-to-one relationship
   ...
end table_text
```

fig. 9.3.   Example subset view component specifications for MViewsER.
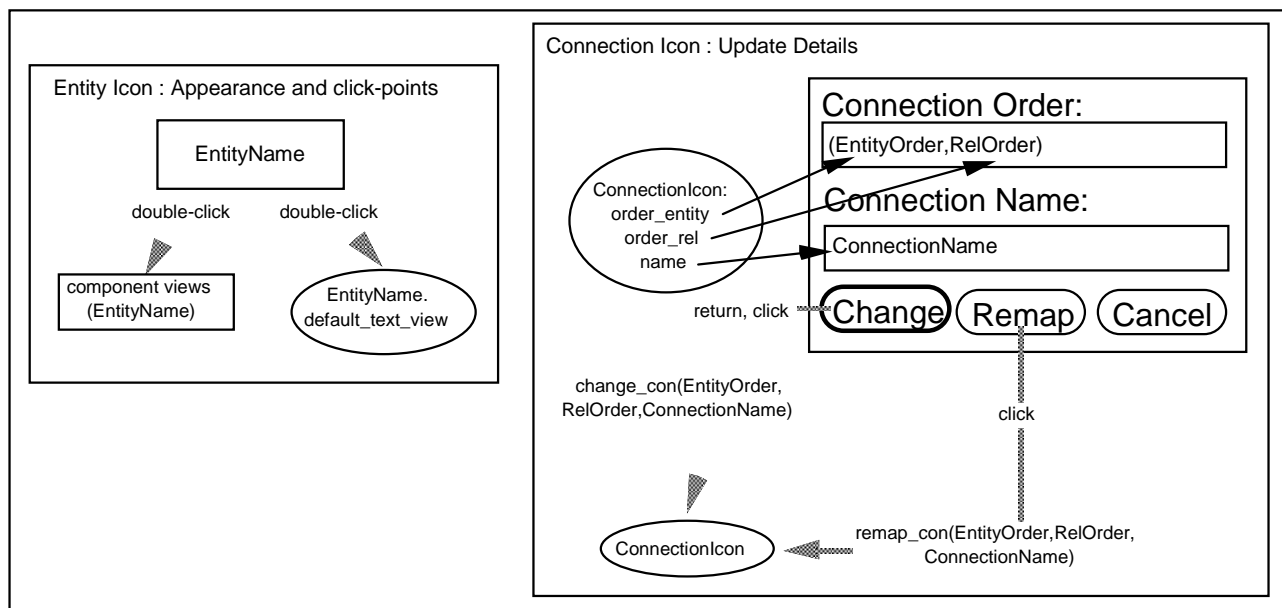
fig. 9.4.   Example display view component and dialogue specifications for MViewsER.

Using the MVSL subset view definitions as a basis, an MVisual specification for display views and dialogues was developed for MViewsER. Fig. 9.4. illustrates some MVisual views defining view component and dialogue appearance and behaviour.

9.1.4. Design

From the MVSL and MVisual specifications for MViewsER an object-oriented design for the environment can be developed by reusing the architecture from Chapter 6. MVSL definitions translate into base and subset component class specialisations while MVisual specifications define the appearance and editing semantics for display views. Fig. 9.5. shows a class hierarchy for MViewsER using the SPE class diagram notation. The MViews architecture classes are abstract (shaded boxes) and the MViewsER classes are on the right-hand side.
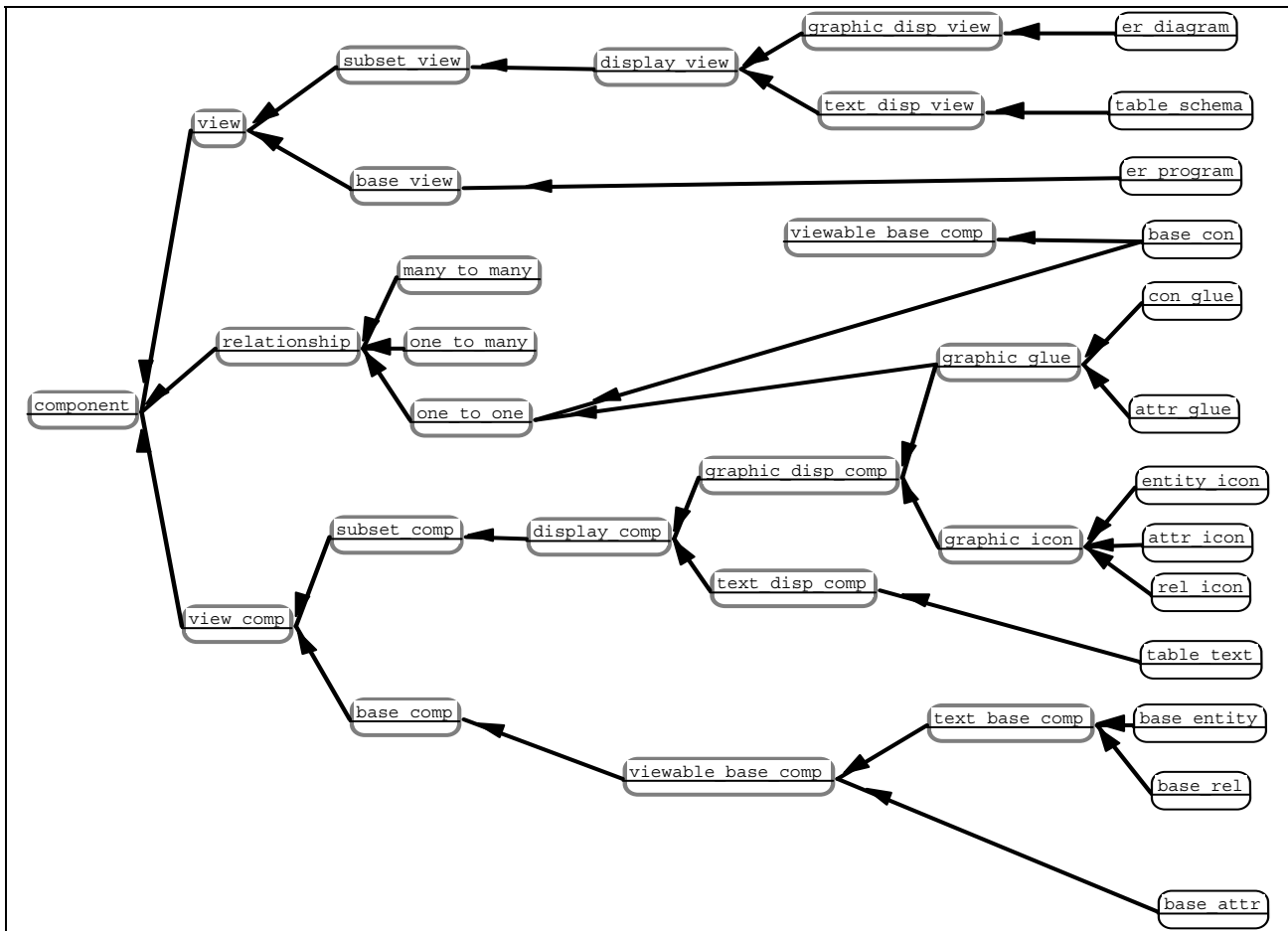
fig. 9.5.   A class hierarchy for MViewsER.

In addition to the classes shown, MViewsER defines subset/base relationships for icon, glue and text form classes and specialises the one_to_many relationship class to represent base component relationships. For example, a base_entity_rels class is defined to relate a base entity to all of its base connections (which each relate the entity to a relationship). These relationship class specialisations implement additional methods to manage the location, compilation, and deletion of components they relate. This approach to modelling MVSL component relationships proved to be much easier to manage and extend than the IspelM approach of using list attributes.

9.1.5. Implementation

MViewsER is implemented by specialising the Snart framework of Chapter 7. The classes defined by the MViewsER design are implemented as Snart classes which inherit from MViews framework classes. MViewsER uses the Snart framework's persistency methods to save and load ER programs. Currently MViewsER loads all base program components when a program is re-opened and saves updated components. This could be extended to provide incremental base component loading, in a similar manner to IspelM.

MViewsER uses lazy update record application to implement a "stay with parent icon" behaviour for attribute icons. When a relationship or entity icon is dragged, its attributes

automatically shift their locations to conform to their parents new location. When an attribute icon is dragged, however, its parent is not affected. The `apply_lazy_updates` method for `attr_icon` examines any `shift_location` update records sent to `attr_icon` and determines whether to move the icon or not.

MViewsER does not currently have a relational database interface. It could be extended to support generation of actual relational schema, however, or even to support graphical querying in a similar manner to (Santucci and Sottile 93, Czejdo et al 90).

9.2. Dialogue Painting

9.2.1. User Interface Specification

User interface specification in most programming languages is via a pre-defined set of procedure calls (Apple 85, LPA 92), reusable object-oriented toolkits supporting graphical user interface components (Linton et al 88), or extending the language with constructs for graphical user interface specification (Haarslev and Möller 90). User interface management systems (UIMSs) provide languages specific to graphical user interface specification (Olsen and Dempsey 85) or interactive user interface specification using graphical editing commands (Myers 89, Avrahami et al 89, Brown 91).

As noted by (Linton et al 88) and (Myers 90), both textual specification (via reuse of toolkits or specialised programming languages) and interactive graphical specification of graphical user interface components have advantages and disadvantages. Textual specification allows other programs to make use of these user interface components (including passing them data to output and being given input data) and provides a precise method of specifying constraints and semantics. Interactive specification allows the appearance of user interface components to be specified in a natural manner by placing and sizing components as required.

A combination of these two approaches has been attempted in systems such as FormsVBT (Avrahami et al 89) and Zeus (Brown 91) where textual dialogue specifications are used in one view and a graphical form in another. These approaches provide consistency between each view representation using a common parse-tree and event generation when the parse tree is updated. (Haarslev and Möller 90) note that in this approach the graphical views tend to constrain the textual representation as both must contain the same data. Other UIMS approaches generate code from interactive descriptions which can be extended by programmers to specify behaviour not easily captured in a graphical representation. A major problem occurs when a graphical representation is changed as previously generated code is over-written by new code. This produces a similar consistency problem to CASE tools that generate code when the generated code or CASE diagrams are modified (see Chapter 4).

9.2.2. MViewsDP

MViewsDP is a dialogue painter for specifying Macintosh-style dialogues using LPA MacProlog built-in dialogue predicates. MViewsDP provides a graphical view which allows dialogue components to be interactively added, deleted and modified. This view shows the form a dialogue will have when actually opened and used by LPA MacProlog. One or more textual views are provided to specify additional information about the dialogue. These contain a Prolog predicate defining the dialogue's sub-components and predicates used to set up initial values for dialogue fields, check the validity of entered data, and carry out any processing of entered data for passing back to Prolog predicates which invoke the dialogue. Fig. 9.6. shows an example of MViewsDP views and an executing dialogue defined by MViewsDP.
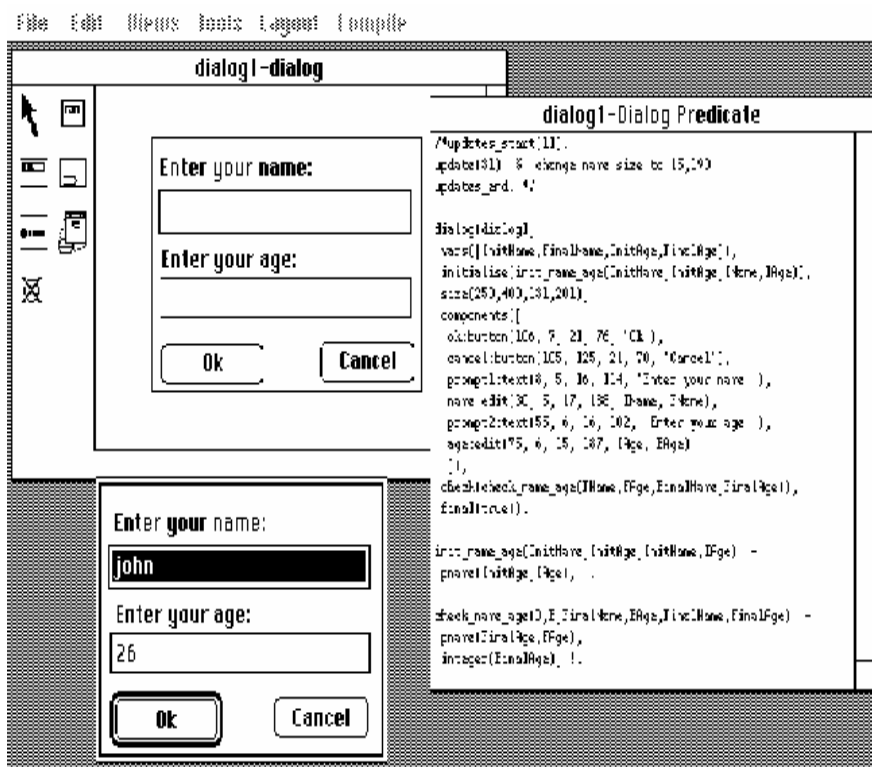
fig. 9.6.   An example of MViewsDP views and an LPA dialogue.

The graphical view has been interactively edited to describe the appearance of the dialogue. The textual view describes the dialogue predicate's Prolog variables (`vars`), its initialisation predicate (`initialise`), its sub-components (`components`), checking predicate (`check`) and final processing predicate (`final`). Dialogue sub-components are named in the textual view by a prefix (unlike LPA dialogue specifications) which helps programmers to determine which dialogue component has been updated. MViewsDP generates a predicate to open the dialogue (an example of an open dialogue is also shown in fig. 9.6.) and asserts other Prolog predicates defined in textual views. Graphical and textual views are kept consistent with update records and an update history is kept. Graphical view

components are redisplayed after receiving updates while textual views unparse updates (as shown in fig. 9.6.).

A difference between MViewsDP and both IspelM and MViewsER is that graphical dialogue sub-components must be enclosed by their owning dialogue's border and are displayed relative to their owning dialogue's location. Sub-component icons are also shifted are resized when their owning dialogue's border is shifted or resized. These dialogue sub-component icons must thus be sub-icons dependent on updates on their owning dialogue. Adding dialogue sub-components to a textual dialogue specification must always result in corresponding graphical sub-component icons in the graphical view (as the graphical dialogue view must contain all dialogue sub-components).

9.2.3. Specification

MViewsDP is specified using MVSL and MVisual in a similar manner to MViewsER. A complication is that a dialogue can contain several sub-components, all of which share common characteristics (location, size and name). A natural way of describing this relationship is to define dialogue sub-components as specialisations of a common MVSL component. To support this MVSL is extended to support generalisation of one component to another. The semantics of this are the same as for object-oriented languages and extended entity-relationship generalisations (Teorey et al 89). A component generalised to another supports all attributes, relationships and operations of its generalisation and can be used whenever its generalisation may be used. Fig. 9.7. illustrates some MVSL declarations for MViewsDP.

```
base element dialogue
   attributes
      name : string
      top : integer
      left : integer
      depth : integer
      width : integer
   relationships
      components : one-to-many dialogue_comp
   ...
end dialogue

base element dialogue_comp
   attributes
      name : string
      top : integer
      left : integer
      depth : integer
      width : integer
   relationships
      dialogue : one-to-one dialogue
   ...
end dialogue_comp

base element text_field
   generalisation dialogue_comp
   attributes
      text : string
   ...
end text_field

base element edit_field
   generalisation dialogue_comp
```

```
   attributes
      initial : string
      final : string
   ...
end edit_field

subset element dialogue_comp_icon
   attributes
      name : like dialogue_comp.name
      top : like dialogue_comp.top
      left : like dialogue_comp.left
      depth : like dialogue_comp.depth
      width : like dialogue_comp.width
   ...
end dialogue_comp_icon

subset element dialogue_icon
   attributes
      ...
   relationships
      components : one-to-many dialogue_comp_icon
   ...
end dialogue_icon

subset element text_icon
   generalisation dialogue_comp_icon
   ...
end text_icon
```

fig. 9.7.   Some MVSL declarations for MViewsDP base components.

An MVisual specification for MViewsDP is used to define the appearance and editing
semantics for display views and dialogues. Fig. 9.8. illustrates some parts of an MVisual
specification for MViewsDP. Dragging a dialogue icon results in all of its sub-component
icons being shifted by the same amount. Sub-component icon details are updated by
dialogues which send events to their MVSL counter-parts.
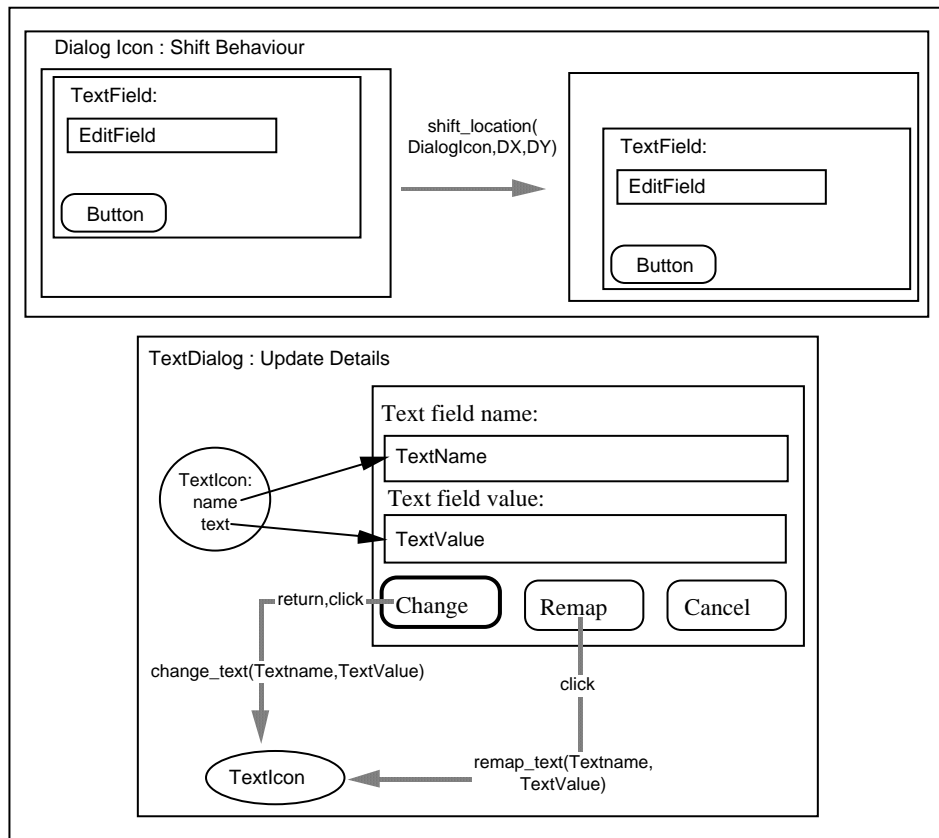
fig. 9.8.   Some parts of an MVisual specification for MViewsDP.

9.2.4. Design

The MVSL and MVisual specifications for MViewsDP are translated into a design by reusing the MViews architecture, in a similar manner to IspelM and MViewsER. Fig. 9.9. shows the inheritance hierarchy for MViewsDP. The MViewsDP classes are shown on the right. `comp_icon` and `comp_icon` are two abstract classes defined by MViewsDP to abstract out common information and behaviour for dialogue sub-components.

`dialog_icon` and `base_dialog` both have a one-to-many relationship to `comp_icon` and `dialog_comp` respectively. These dialogue sub-components are dependents of the dialogue and its icon and hence are sent update records from the dialogue. Dialogue sub-component icons are displayed relative to their owning dialogue icon's position. Hence they must shift their location when the dialogue icon is moved or resized and must ensure they do not overlap the dialogue icon's border.

9.2.5. Implementation

MViewsDP is implemented by specialising the Snart framework and basing the new Snart classes on those defined in the design for MViewsDP. As with MViewsER entities and relationships, `base_dialog` and `dialog_icon` use relationship components to manage their relationships to instances of `comp_icon` and `dialog_comp` specialisations.

MViewsDP uses the Snart persistency mechanism to save dialogue program states. This proved to be a much more abstract way of handling incremental program saving and loading than the MViews framework `save` and `load` methods. The MViewsDP classes are persistent classes and do not have to provide any other facilities to save and reload their states. This contrasts dramatically to the programming effort involved in supporting incremental saving and loading in IspelM and simple program saving and loading in MViewsER. For MViewsER, a significant portion of its development time was spent defining what information for each component class needed to be saved and how to restore it. Effort must also be expended to specify how this persistent information is deleted when the component is deleted (for example, deleting an appropriate resource from a save file). The MViewsDP approach using Snart object persistency proved to be a much better way of handling program persistency. Chapter 10 draws on this experience to develop an improved model for component persistency which also supports version control and multi-user distributed programming.
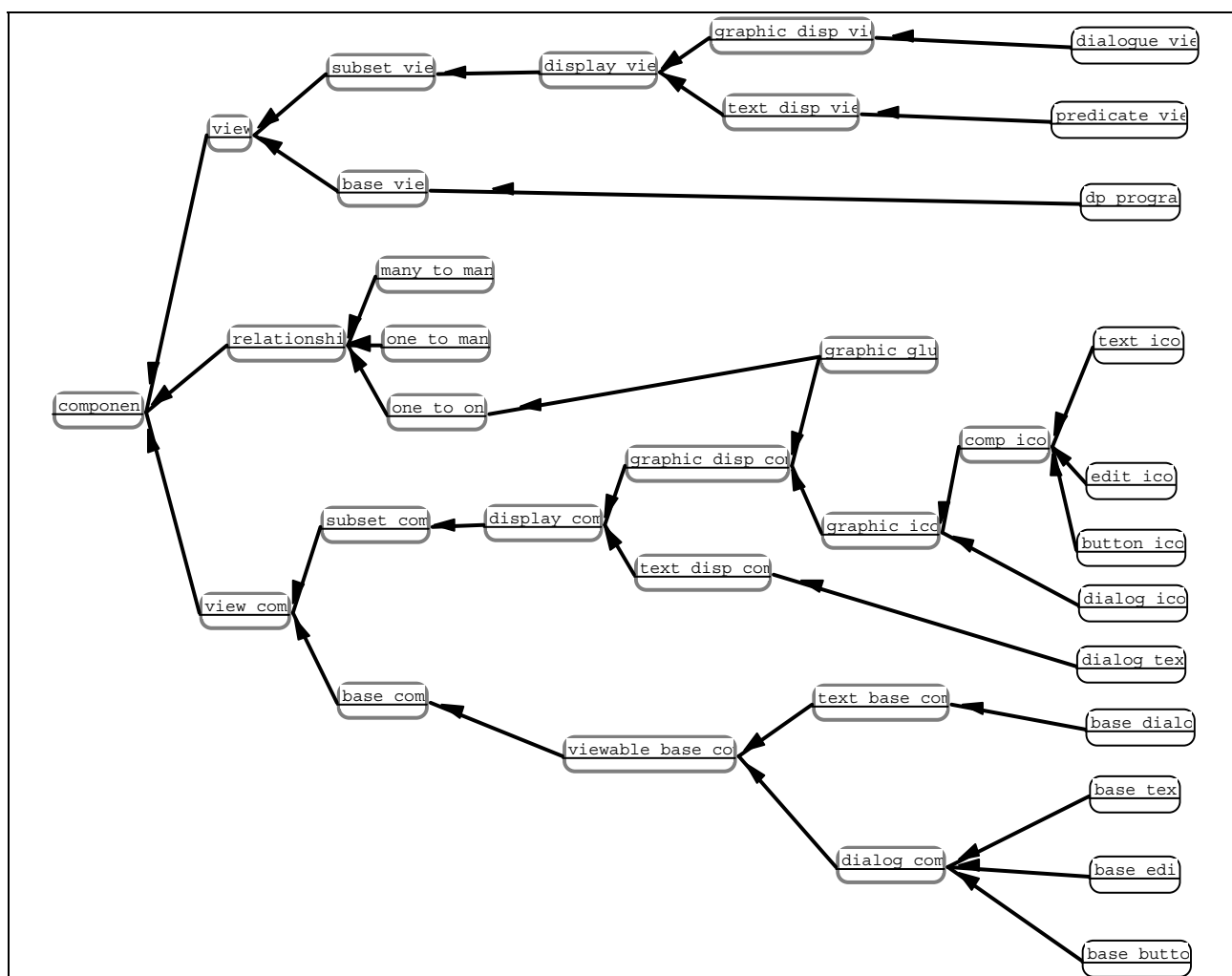
fig. 9.9.   Class hierarchy for MViewsDP.

Lazy update record processing was used to support `comp_icon` and `dialog_comp` update record processing. `comp_icon` needs to determine whether `shift_location(Dialog, DX,`

DY), `update_attribute(Dialog, depth, NewDepth)` and `update_attribute(Dialog, width, NewWidth)` have been sent to a dialogue sub-component icon from its owning dialogue. The dialogue sub-component icon can then reconfigure its state to conform to that of its owning dialogue (if necessary).

Another interesting use of lazy update record processing is by `dialog_comp`. Changing the position and size of dialogue sub-components is a common task which generates many update records (typically four if the top, left, depth and width attributes of `dialog_comp` are all updated by the one editing operation). dialog_comp uses lazy update record processing to merge these updates into one update record of the form `change_comp(SubComp, OldTop, OldLeft, OldDepth, OldWidth, NewTop, NewLeft, NewDepth, NewWidth)`. If only size or location is updated, update records of the form `change_size(SubComp, OldTop, OldLeft, NewTop, NewLeft)` and `change_size(SubComp, OldDepth, OldWidth, NewDepth, NewWidth)` are generated. This helps to reduce the number of update records generated and hence assists programmers by reducing the number of update records stored and expanded in textual views. Chapter 10 discusses ways MViews can be extended to better facilitate and automate this kind of update record composition.

9.3. Program Visualisation

Program visualisation systems allow programmers to see parts of their programs in an abstract (possibly graphical) manner to help facilitate understanding and/or debugging of programs (Meyer 90). In this section three dynamic[30] program visualisation systems developed using MViews are discussed. The first two are simple examples illustrating some of the diverse applications MViews can be used for. The third is a visual debugger for SPE being developed at the University of Auckland by Stephen Fenwick, based on a prototype system that did not use MViews (Fenwick and Hosking 93).

All of these systems use the Snart object spying mechanism (described in Chapter 3 and Appendix B) to generate low-level tracing events on objects. MViews converts these events into update records which are propagated to subset and display view components to drive animations and maintain subset and display view consistency with an executing program.

---

[30]Dynamic program visualisation displays views of the execution state of a program. SPE supports static program visualisation by allowing programmers to construct class diagram views illustrating the structure of programs.

9.3.1. Tally Graph of Method Calls

A tally graph of method calls to an object illustrates the amount of usage of individual methods (Noble and Groves 92). Fig. 9.10. shows such a tally graph for a `drawing_window` object from the drawing program described in Chapter 4. This bar graph view shows a count of the method calls to the `drawing_window` object for an instance of the drawing program. As the `drawing_window` object's methods are called this tally graph is updated dynamically.
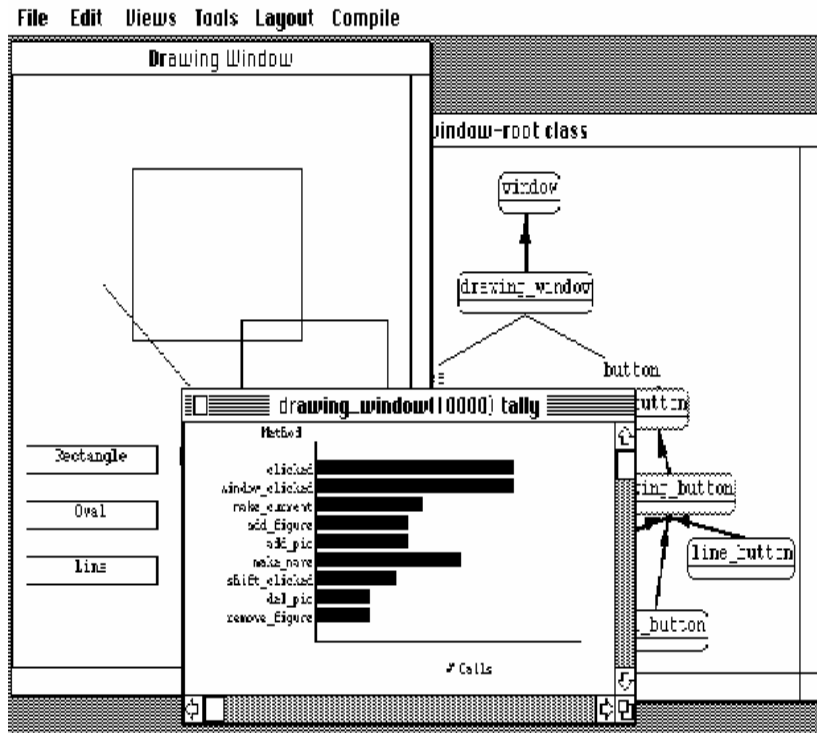
fig. 9.10. An object method call tally graph for the drawing program.

The tally graph view is implemented as an extension to SPE. A user-specified object's entire interface is spied by a Snart predicate call of the form `sn_trace_object(Object)`. The `spe_program` object sends `sn_entry(Object, Method(ArgumentList))` events generated by the spied object to a `hashtable` base component as update records. This base hashtable converts these update records into `insert_item(Method,1)` or `update_item(Method,NumCalls+1)` method calls on itself. The hashtable is an "active" data structure in that it inherits from `component` and generates update records when it is changed.

The hashtable has a bar graph subset component which is specialised to a bar graph display icon (which draws the axes shown in fig. 9.10.). The bar graph subset component is connected to the base hashtable by a subset/base relationship. This subset/base relationship translates hashtable update records into bar graph subset component method calls by over-riding the `subset_rel::update_from_base` method. The bar graph display is redrawn when it receives `update_display` calls from its subset component. Each bar graph

bar is a sub-icon of the bar graph display icon and renders bar graph label and bar pictures. A bar graph display view is used to enclose the bar graph display component in a window. Fig. 9.11. shows the translation of object events into update records and bar graph updates.

The base hashtable used by the tally graph view could have more than one subset view with different kinds of display views. For example, a textual view might print out a method call trace with the argument values for each method call (if the base hashtable propagates the `sn_entry` events sent to it as update records). The base hashtable might also store update records against itself to document all method calls sent to the Snart object.
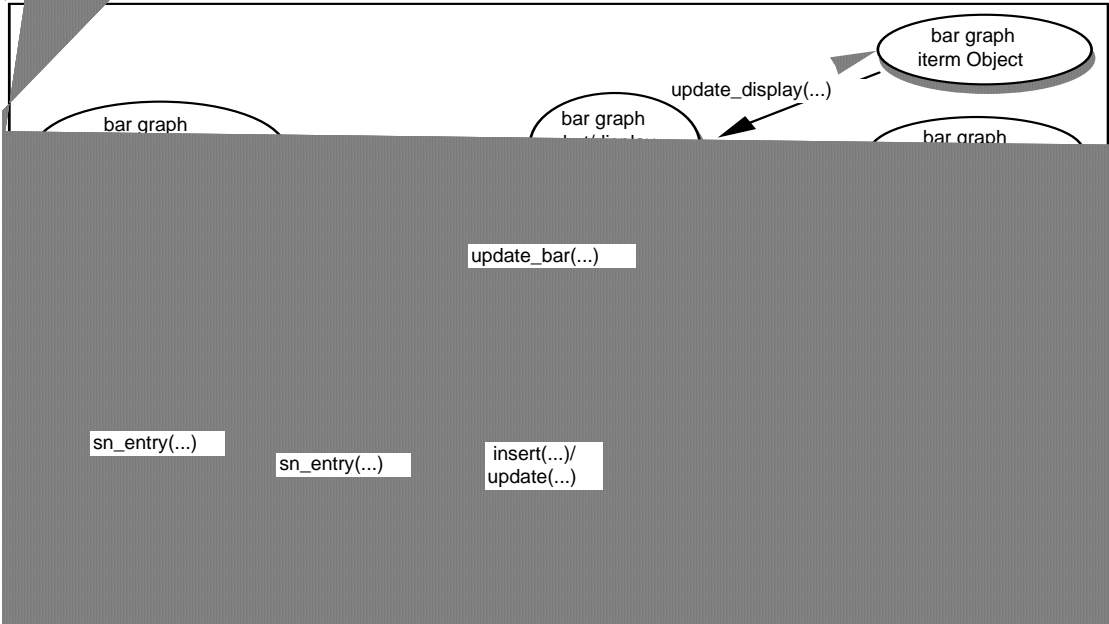
fig. 9.11. Spied object events and update record propagation for the method call tally view.

9.3.2. Sorting Algorithm Animation

Algorithm animation systems attempt to illustrate how an algorithm works by visually demonstrating interesting events and corresponding data and control flow modifications that occur during the algorithm's execution (Brown 88, Stasko 89, Myers 90). Sorting algorithms are a common example used. A graphical view illustrates how and when data is compared and moved during execution of the sorting algorithm. Some animations also show the commands being executed at each step of the algorithm. Such animations are useful both for teaching how sorting algorithms work and for testing algorithms for correctness (an error will result in an in-correctly sorted data structure) and efficiency (an in-efficient or erroneous algorithm may produce an unduly long or incorrectly sequenced animation).

A simple sorting algorithm animator has been implemented as an extension to SPE. Fig. 9.12. shows an example of the sorting animation view during execution. As the sorting

algorithm compares two values, their bars are highlighted, and when two item's values are swapped, the bars are exchanged. This example shows the progression of a bubble-sort algorithm sorting elements in a list.

To implement the sorting animation view, a sorting algorithm object's `compare(Item1, Item2)` and `swap(Item1, Item2)` methods, or a data structure's `set_item(Item, Value)` method, are spyed to generate "interesting events" (Brown 88) which drive the animation. `sn_entry` events are sent to the `spe_program` object which then propagates the events (as update records) to a subset/base relationship (with no actual base component - the spied object is assumed to be the "base" component). This subset/base relationship uses exactly the same bar graph subset and display components and views as used for the tally graph view to display an animation of the sorting algorithm. The subset/base relationship converts `sn_entry(Object, compare(...))`, `sn_entry(Object, swap(...))` and `sn_entry(Object, set_item(...))` update records into appropriate bar graph subset manipulation methods.
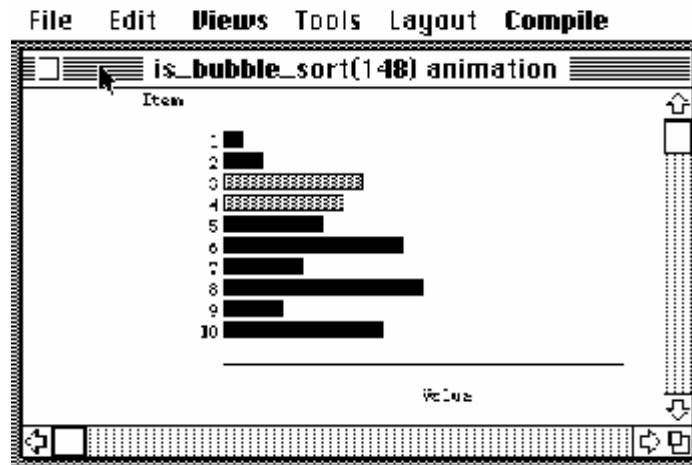
fig. 9.12. An example of a sorting animation view.

9.3.3. Visual Debugging

Visual debugging allows executing programs to be debugged using graphical diagrams describing object data, relationships and control flow (Fenwick and Hosking 93). Cerno, a visual debugging system for SPE, is under development at the University of Auckland and reuses MViews to build graphical debugging views for Snart programs (Fenwick 94).
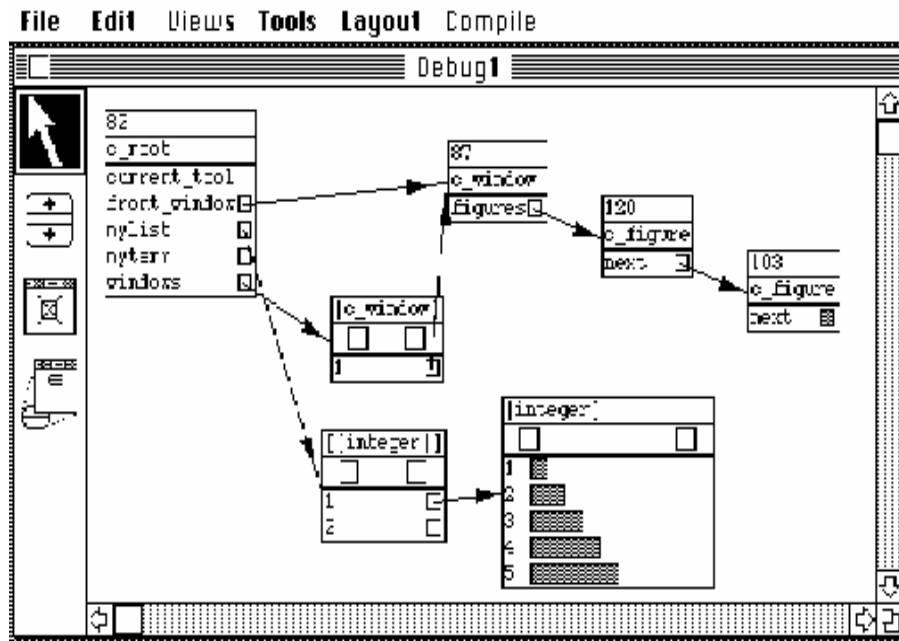
fig. 9.13. An example of Cerno multi-object views.

Cerno defines views and view components to describe the state of an executing Snart program. Viewed Snart objects have their methods and attributes spied so debugging views can be updated when the objects change. This ensures view consistency with the executing Snart program (i.e. the "base" view). A multi-object view shows one or more Snart objects and references to other Snart objects. Programmers can specify which attributes of an object are shown for each object view component. References are expanded by programmers and object attribute values can be updated. Fig. 9.13. shows an example of Cerno multi-object views.

Cerno is implemented by specialising the MViews Snart framework from Chapter 7. Subset component classes are defined to represent Snart objects, list attribute values and term attribute values. Corresponding subset/base relationships are sent update records by the base view when the objects they view are updated by Snart. These update records are translated into changes to subset components which are then re-rendered to reflect the changes to the Snart objects.

Updates on Prolog lists and terms need to be handled by different subset components as Snart does not store them as objects. A list attribute subset component is a dependent of an object subset component. It is sent update records received by the object subset component from a spied Snart object. Any update records describing changes to the list attribute are converted into changes to the list attribute subset component and it redraws its display to reflect this change. Terms are viewed as lists of a fixed length by converting a Prolog term to a list using the =.. operator.

Object subset components can be rendered in a variety of ways, as shown in fig. 9.13. This allows an object to display information about its attribute types (as a large object display

component) or as a small form showing user-specified attributes and object references. Snart classification is used to dynamically select between different types of display components (for example, bar graphs or lists) depending on a programmer's preference for how an object's data should be displayed. Further details about Cerno and its implementation can be found in (Fenwick 94).

9.4. Other Applications of MViews

This section briefly discusses further applications of MViews currently under development or planned for development. These systems are quite diverse in nature and illustrate how the MViews notions of object dependency, update records, and multiple textual and graphical views with consistency management can be usefully reused in different systems.

9.4.1. Facets and Object Persistency for ICAtect

The ICAtect system (Amor et al 91) defines a Common Building Model for modelling building designs. This model specifies classes of building components, attributes for these components and relationships between these components. Instances of this model can be constructed which define a particular building design. Existing CAD and engineering tools can be interfaced to ICAtect and can thus use the Common Building Model for data interchange. An initial prototype of ICAtect was implemented in Prolog and provided a mainly textual interface (Amor 91).

A new version of ICAtect is being developed at the University of Auckland by Robert Amor. This reuses SPE to model and construct a Common Building Model using SPE's class diagrams and the model is stored as Snart classes. Instances of the model are constructed by using a derivative of Cerno to view and manipulate building designs. The Snart persistency mechanism allows these instances to be automatically made persistent by Snart.

Snart has been extended to incorporate *facets*. A facet is a named value associated with a class attribute and is used to specify additional information about the attribute, such as default values, a description, and type and constraint information[31]. For example, an attribute might be declared as:

```
shape(facets([
  relationship(values),
  type(shape),
```

---

[31]A facet could be compared to a relational database table field value which specifies similar kinds of information for table fields.

```
        constraints(instanceof([shape])),
        description('The shape of this particular object, defined in the hybrid
        edge data structure, see Y. E. Kalay in Computer Aided Design, Vol 21,
        No. 3, April 1989')
        ])),
```

SPE and Cerno have been extended to incorporate this notion allowing facets to be visually represented and manipulated. This provides a programmer-level visualisation and visual programming interface for ICAtect with most facilities implemented by SPE and Cerno. Snart objects produce a much faster performing building design database than the original Prolog database terms used to define the Common Building Model and its instances. SPE and Snart object persistency allow the Common Building Model and its instances to be automatically and incrementally saved and reloaded.

9.4.2. Support for More OOA/OOD Notations

Extending SPE to incorporate more extensive design and analysis views would provide an improved environment for such high-level software development tasks. Of particular interest are graphical views which allow control and data flow between class features to be specified (similar to those described in (Fichman and Kemerer 92)), textual class interface contracts to be defined (for defining class contracts as used by (Meyer 92)), and analysis to design (and vice-versa) consistency management.

Fig. 9.14. shows what examples of such views might look like. The top view is a feature control and data flow view illustrating some feature calls between the `figure` and `drawing_window` classes. This view also shows some of the arguments passed between features including argument names, types, and input and output arguments. Some calls are sequenced to show the order they occur from a method (for example, `drawing_window::clicked` calls `figure::pt_in_figure` and then `figure::select`). This kind of design-level view is useful for specifying detailed control and data flow between individual class features (Fichman and Kemerer 92). The existing SPE class diagrams can not capture this level of detail. Other facilities of this type of view might include hypertext-like browsing capabilities which display method code views when a connection is double-clicked on.

The class contract textual view defines high-level pre- and post-conditions associated with methods (of a similar style to those of Eiffel (Meyer 92) but using Prolog clauses to specify validity conditions). It also defines the arguments supported by methods (and could define argument types as well).

Both the feature control and data flow view and class contract view must be kept consistent with changes to other SPE views. Changes to these views must also be propagated to other graphical and textual views which share information displayed in

these views. Analysis and design views can be kept consistent with each other (for example an abstract class diagram view and a feature control flow view) by update records propagated between base components. For example, a change to an abstract client-supplier relationship might affect a related feature control flow relationship. The feature control flow can be made a dependent of the client-supplier and be sent update records. These records can be used to update the control flow relationship's state or be stored so programmers are informed that it has been affected by a change to the client-supplier.

To extend SPE to support these (and other) analysis and design views, the MViews model, architecture and framework can again be reused. Base classes and features would be extended to support relationships to new base components representing method call relationships, method arguments, and method pre- and post-conditions (possibly stored as textual base components).

For feature control flow views `class_icon` could be specialised to a new class icon which supports feature sub-icons. Dragging a class icon would shift its feature sub-icons while shifting feature sub-icons would alter their position in relation to the owning class icon (implemented in the same way as MViewsDP dialogue icons and dialogue sub-component icons). Feature control and data flow connections would be `graphic_glue` specialisations with sub-icons representing arguments and an attribute indicating feature call sequencing.

For class contract views `class_text` could be specialised so parsing a pre- or post-condition updates the appropriate base component's text form. MViews supports copying text form data from part of a textual display view and, as the key-words `before` and `after` are used to distinguish contracts (see fig 9.14.), these could also be used to locate the text defined by pre- and post-conditions. Class contract text forms could be extended to support partial class interface displays with inherited features and conditions, similar to SPE's class code text forms (see Chapter 4).
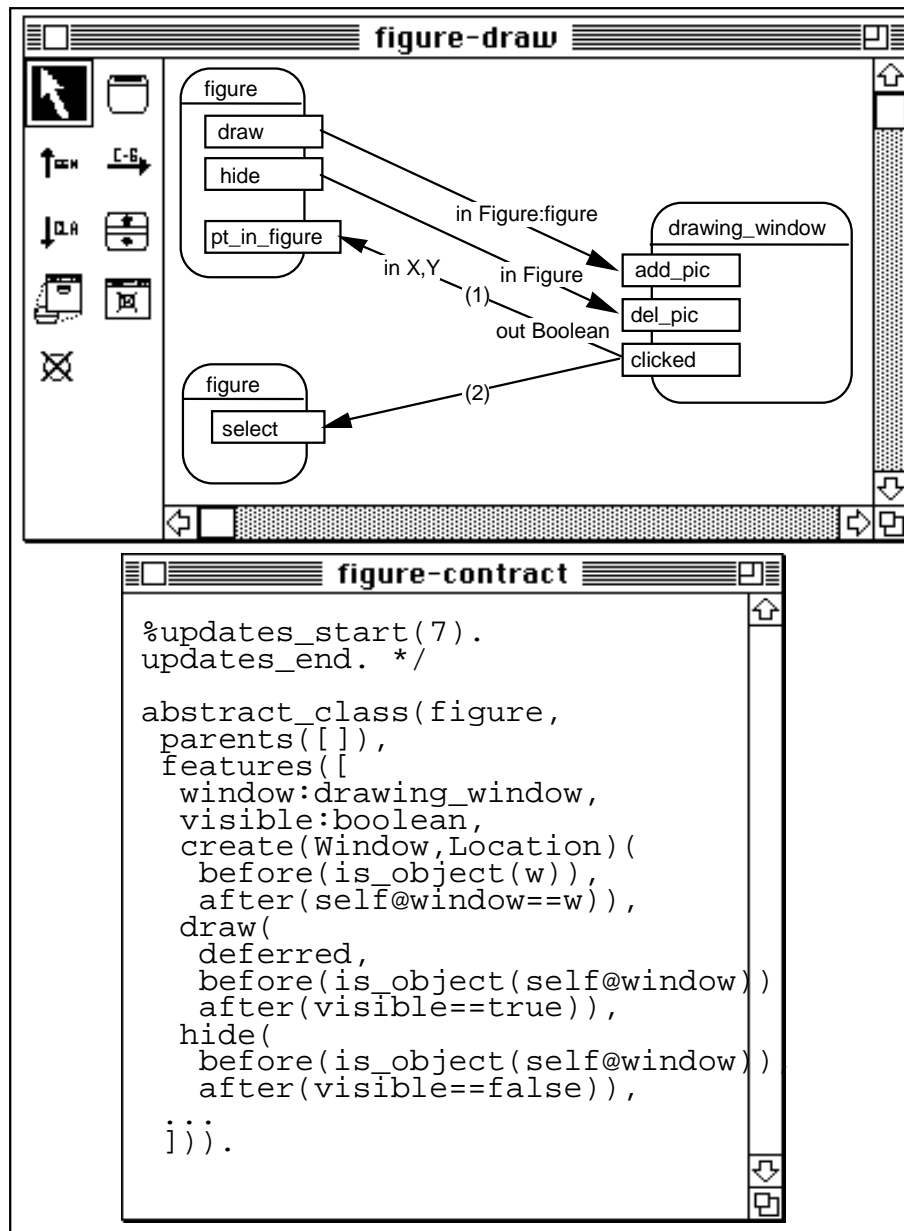
fig. 9.14. Examples of feature control flow and class contract views for SPE.

9.4.3. Dataflow Analysis Diagrams and Method Implementations

Dataflow diagrams (DFDs) are useful for systems analysis where they show high-level dataflow between entities or classes and objects. DFDs can also be used for visual programming. Low-level dataflow is specified between "boxes" along "wires", and boxes can either represent fundamental operations or be composed of other boxes and wires. Methodologies using DFDs for analysis include Bailin Object-Oriented Requirements Specification (Bailin 89) and Shlaer and Mellor Object-Oriented Analysis (Shlaer and Mellor 88). Visual programming systems using DFDs include Fabrik (Ingalls et al 88) and Prograph (Cox et al 89).

SPE could be extended to provide DFDs for both analysis and visual programming. Fig. 9.15. shows an analysis-level DFD view and a method DFD view. Analysis-level DFDs

describe classes and methods (rather than disembodied processes, as used by conventional DFDs for structured analysis (Fichman and Kemerer 92)), and show dataflow connections (possibly named and including an indication of the data passed) between classes and methods. Method DFDs show operations in boxes (which may be other class methods, Prolog predicates or other DFDs) and connections (possibly named) between operation boxes. Shaded wires indicate synchronisation of operations. This style of DFD program is based on that used by Prograph (Cox et al 89), with additional support for boxes that interface to textually implemented methods and Prolog predicates.
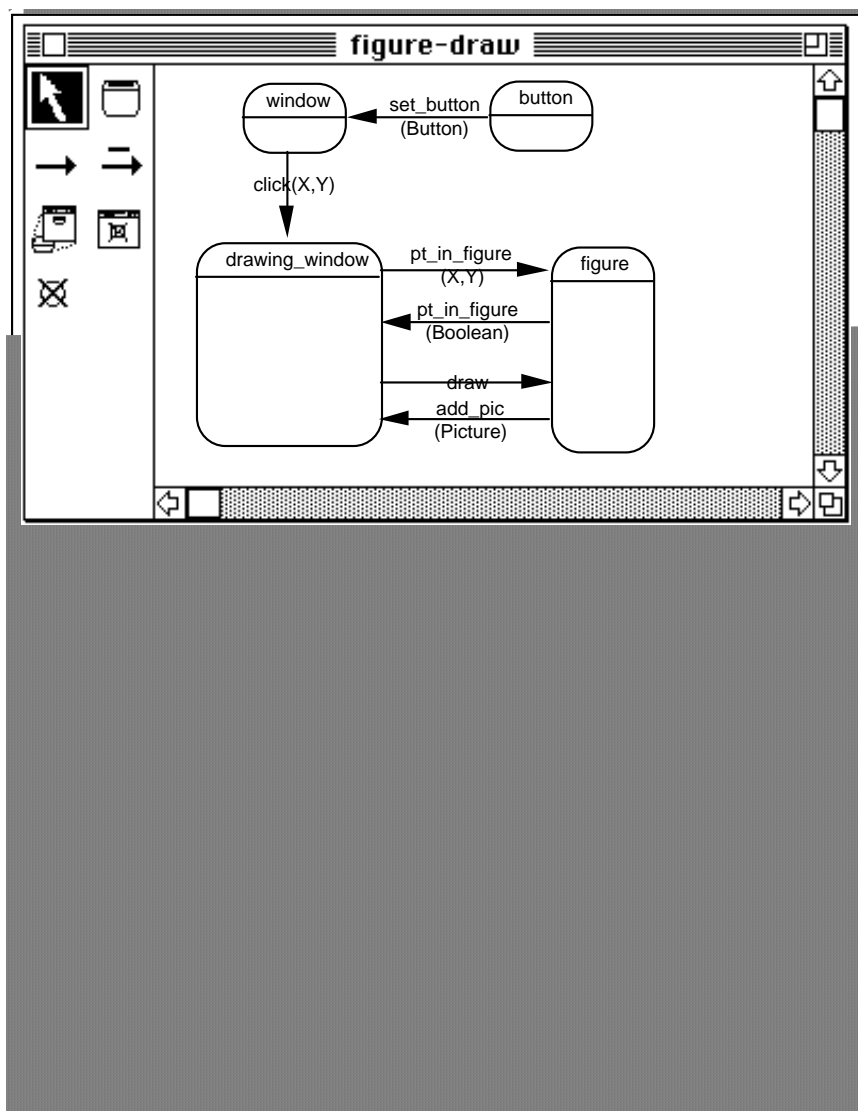
fig. 9.15. Examples of analysis-level DFD views and method DFD views for SPE.

To implement DFD views in SPE, base class and feature components must be extended to incorporate dataflow relationships (for analysis DFDs) and dataflow box definitions (for method DFDs). Analysis dataflow relationships require names and base components for representing data passed via the dataflow connection. Method DFDs require base component boxes and pins to represent the external structure of a box. They also require a method of specifying the internal structure of boxes using other box interfaces (boxes and pins) and wires between box pins. This internal structure could be specified in the base

view or, as box definitions are hierarchical, the structure of a display view could be used to define both the appearance and internal structure of a box.

Analysis DFD views can specialise the existing `class_icon` class and support dataflow connection glue (with optional names and data values represented). Method DFD views require box icons and pin sub-icons and wire glue. Analysis DFD diagrams can use update records to keep them consistent with other views and can provide hypertext-like capabilities for moving to other SPE views. Method DFD base component boxes must be kept consistent under change. This includes updating a box's internal specification using update records (whether its specification is a DFD, Prolog predicate or textual method definition) when its external interface is changed. DFD internal specifications are hierarchical and thus can be updated automatically (by adding, moving or deleting a box's pins shown when the box is used in other method DFD views). Textual predicates or methods used as boxes in method DFD views must be updated by expanding update records into their textual display views. For example, if a DFD view renames wires or adds pins to a box representing a Prolog predicate, the predicate's textual definition, currently defined in a class or feature text form, must have update records expanded in the text form to reflect the change. Examples of such unparsed update records are shown in fig. 9.16.

```
/* updates_start(7).
update(1). % rename pin (argument) Picture to LPAPicture
update(2). % add unnamed input pin
update(3). % remove output pin 1
updates_end. */

drawing_window::add_pic(Window,Picture,Drawn) :-
   ...
```

fig. 9.16. Unparsing update records from modified boxes into a textual method implementation.

Fabrik and Prograph allow executing DFD programs to be displayed using the views defined to construct the program (Fabrik's diagrams are viewed as always executing, even for partially constructed programs). SPE method DFDs could support such a facility if MViews component classes are defined to hold the run-time execution state of boxes. Such run-time base components could duplicate the composition and layout of method DFD views to allow programmers to view the state of executing box pins. An alternative might be to use Snart objects to store the state of an executing DFD box and spy these objects so an SPE method DFD view can be used to browse the executing program's state.

9.4.4. Tool-based Abstraction

Garlan et al propose using tool abstraction to support the evolution of large-scale software systems (Garlan et al 92). They compare and contrast using abstract data structures and tool abstraction as the basic modelling technique for such systems. Tool abstraction involves composing a system from reusable "tools" which supply data processing. Tools share a set of abstract data structures which they update and data can "flow" between tools for different processing. Garlan et al claim that while data abstraction eases design changes for data representation, tool abstraction does the same for system functionality.

Two requirements for tool abstraction are: some method of determining whether data structures have changed; and on data structure change, tools dependent on the data structure state must be triggered so they can process data and maintain a consistent system state (Garlan et al 92). One approach to driving this tool invocation process is by utilising "active" data structures. Some object-oriented systems providing such facilities include Smalltalk (Goldberg and Robson 84), Flavors (Moon 86) and many object-oriented databases (Garlan et al 92). A problem with most approaches is similar to that for view consistency discussed in Chapter 5: only an indication of some change or an indication of some object attribute change is given to dependent objects (tools), rather than the actual change that occured.

The MViews framework can be reused to support tool abstraction by using "active" data structures which generate update records documenting the exact change a data structure has undergone. Tools can also be implemented as component classes which are dependent on various active data structures. Tools may even communicate via generation of update records in contrast to explicit method invocation and thus may be dependents of one another.

Fig. 9.17. shows an example (based on one from (Garlan et al 92)) of tool abstraction using specialisations of MViews `component` class. In this example: an input tool reads lines from a file and inserts them into a shared line buffer (an active list); a circular shifter tool appends the first word from each line to the end of the line and stores them in a shifted line active list; an alphabetiser tool orders this shifted line list; and an output tool writes the sorted, shifted lines to a file. Fig. 9.17. shows the update records generated by each data structure, which are used to drive this line shifting process, and the method calls on data structures by tools. Tool classes can be specialised and new tools added by changing the dependencies in the system (for example, to include an omit tool for removing blank lines from the shifted line buffer before output (Garlan et al 92)).
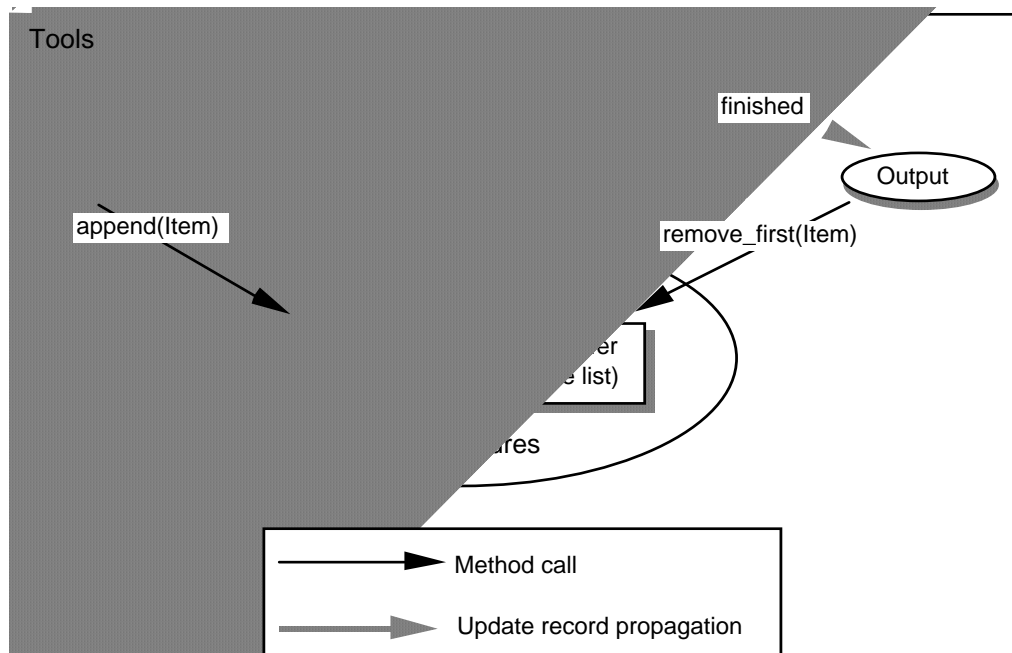
fig. 9.17. Active data structures and tools supporting tool abstraction.

The MViews model alleviates some of the problems of tool abstraction discussed in (Garlan et al 92). These include: explicit tool invocation (tools are invoked as required by MViews' object dependency mechanism); inefficient response to data changes (tools are sent an update record describing exact data structure change and thus can provide specialised, efficient processing, in the same manner as subset view updating (see Chapter 5)); lazy vs. eager data processing (tools can schedule processing of update records lazily using MViews' lazy update record processing); and tool scheduling (tools can be dependent on one another and thus a tool can wait until another tool has processed data before it processes data).

9.5. Discussion and Future Research

9.5.1. MViewsER

MViewsER provides graphical ER diagram views with corresponding RDS textual views. Update records are used to propagate changes between these interactively edited ER diagrams and free-edited and parsed textual RDSs. Unlike most ER diagram/RDS systems, MViewsER propagates all changes affecting entities and relationships to RDS views so programmers can determine whether they affect the RDS definitions. For example, changing the cardinality of a relationship connection may affect field values (such as defaults, descriptions and ranges) for both the connected entity and relationship RDSs. Consistency management is employed to always keep renamed, added or deleted entities, relationships and their attributes, and their corresponding RDS tables and fields, consistent.

MViewsER can be extended to allow programmers to explicitly define normalisation of entities and relationships (Teorey et al 86). Entities and relationships can be implemented as relational database tables or can be implicitly defined by table fields. For example, an `account-of` relationship between `customer` and `account` entities might be zero-to-many from `customer` to `account` and one-to-one from `account` to `customer`. A relational database might store such relationships as a `customer_id` field in the `account` table, rather than have a separate `customer-of` table with `customer_id` and `account_id` fields. A relational database query can find all accounts for a given `customer_id` value, ID, by a simple query of the form:

```
select account_id from account
    where account.customer_id = ID
```

Extended entity-relationship (EER) modelling, as used by (Teorey et al 89, Czejdo et al 90) allows generalisation relationships to be defined between entities. For example, a `business_customer` entity might be a specialisation of `customer`. MViewsER could support such relationships, and mutual inclusively and mutual exclusivity constraints between specialisation entities, by providing base generalisation relationships and graphic generalisation glue.

(Teorey et al 89), (Czejdo et al 90) and (Santucci and Sottile 93) describe systems which support graphical construction of ER queries. MViewsER includes relational database definitions as well as ER diagrams and thus could, in theory, support more explicit and efficient relational database graphical queries if MViewsER supported programmer-defined normalisation of entities and relationships, as discussed above. A graphical ER query on the `customer` entity requesting all accounts for a customer id could use the RDS for `customer` and `account` to determine the database select query shown above. This contrasts with some systems which apparently assume an ER model with both entities and relationships implemented as relational database tables, such as (Czejdo et al 90) and (Santucci and Sottile 93).

9.5.2. MViewsDP

MViewsDP provides a graphical dialogue painter view and one or more textual dialogue and dialogue predicate views. This allows the appearance and layout of dialogues to be interactively specified while dialogue constraints, input and output variables, and defaulting and data conversion to be managed textually. Textual dialogue predicates are asserted directly as LPA predicates while textual dialogue specifications are translated into a predicate which implements dialogue data initialisation, dialogue opening, data validation and final data conversion and return. In contrast to most interface building systems, MViewsER allows different levels of detail to be supported naturally in each view, keeps most aspects of graphical and textual dialogue specifications consistent, and

indicates to programmers changes that can not be automatically applied to a dialogue view.

MViewsDP currently provides button, text field and edit field dialogue sub-components. This simple dialogue model could be extended to include radios, check-boxes, menu selections and pop-up menu items (perhaps using the Lean Cuisine notation of (Apperley and Spence 88)) so a more complete range of dialogues can be specified. MViewsDP could also inform programmers of invalid dialogue formats by visual displays. For example, a dialogue item whose border overlaps that of its owning dialogue could be rendered in red to illustrate an error. Similarly, LPA MacProlog requires dialogue edit fields to have valid input and output values (inputs must be non-variables and outputs can specify various "reading mode" terms, such as gread(Variable) and tokens(TokenList)). MViewsDP should ensure all dialogue sub-component definitions are correct before attempting to generate LPA dialogue predicates. Errors could be reported using update records, as used by SPE for compilation and semantic errors.

A generalised form of MViewsDP could be used to visually and textually specify more general graphical user interfaces. MVisual demonstrated that visually specifying the appearance and editing functionality of MViews views and view components is a natural and expressive approach. An extended MViewsDP could be used to generate icon, glue and view appearances (and possibly some functionality) using similar graphical and textual views to dialogue specification views.

9.5.3. Program Visualisation

Program visualisation systems can be built using MViews and Snart's object spying mechanism. A tally graph view indicates a count of method calls to individual object methods. A sorting algorithm animator displays each compare and swap step for sorting algorithms. A visual debugger provides graphical multi-object views showing the state of objects and their relationship to other objects.

While there are no plans to extend program animation views for SPE, the visual debugger is currently being extended to provide control flow visualisation between object methods and to support generation of multi-object views from SPE class diagrams and vice-versa. Control flow between objects can be visualised by spying all methods of objects viewed in a multi-object view. When a spied object calls another spied object's method, the called object generates events which can be used to graphically illustrate the current method call (and method call sequencing). Arguments to method calls could be displayed when requested by programmers or by default in a multi-object view.

Generating multi-object views from class diagrams (and vice-versa) will allow a "schema" for multi-object views to be reused and saved with an SPE program. Update records will

be used to keep schema and object views consistent under change. Spied object events can be stored as update records against object icons to provide a history of attribute updates and method calls on an object. This would allow programmers to review the "update history" for objects and thus assist in locating incorrect attribute value assignments and incorrect method calling sequences.

9.5.4. MViews

Development of IspelM and the systems described in this chapter have indicated good and bad aspects of the MViews model, its specification languages, architecture and framework. MViews greatly reduced the time taken, in comparison to using raw LPA MacProlog, to model, design and implement the systems described in this chapter. For example, MViewsER and MViewsDP took less than a person week each to develop from initial specification with MVSL and MVisual to final implementation using the Snart framework. The method tally view and sorting animation views took less than a day each to design and implement. Use of MViews for the development of Cerno has lead to a much faster development time, less errors during implementation and much improved functionality and extensibility than with the original Cerno prototype which did not use MViews (Fenwick 93 and 94). Similarly, development of IspelM and SPE has been greatly enhanced, in terms of development time, extensibility and useful functionality, compared with the original Ispel system (Grundy et al 91, Grundy and Hosking 93).

Of particular benefit when developing these environments has been:
- MViews' model and specialisable component class hierarchy for defining base view components and subset views and components for different systems. As this is graph-based and stores both language structure and semantics it has proved flexible for many diverse applications. Abstractions such as viewable base components, graphical icons and glue, and graphical and textual display views allows specialised components to be defined which inherit a large amount of useful functionality (which is also consistent with other specialised component state and behaviour).
- MViews' object dependency model using update records is useful for propagating changes between dependent components, maintaining view consistency, indicating changes affecting detailed textual views, and documenting component changes. A generic undo/redo facility and lazy update record processing has supported quite diverse use of update records including maintaining visual layouts and constraints and maintaining component state consistency. The homogenous nature of MViews, with all these facilities based on object dependency graphs using relationship components and update record propagation, has made reuse of MViews very straightforward.

- MVSL and MVisual for initial environment specification using the MViews model. MVSL proved useful in the design of MViewsER and MViewsDP for defining important component structures, operations and update responses. These basic environment abstractions could be reasoned with at an abstract level using their MVSL specifications. MVisual allowed the appearance and basic functionality of display views for these environments to be defined for MVSL subset views. MVSL and MVisual specifications do not capture all of the information needed to define a new environment, but proved very useful for initial design of an environment and for determining how subset and display views interact.

- The MViews architecture and framework allowed an MVSL/MVisual specification for an environment to be translated into an implementable design and then a specialised framework of classes. Environment design refined MVSL attributes and relationships to describe how they will be implemented, defined subset/base relationships to translate between base and subset component updates, and described MVSL subset views and components and MVSL display views and components using one class of object. The framework refined this design to describe exactly how components respond to update records using declarative methods, implemented relationships and attributes using Snart class attributes, and provided a persistency mechanism and user interface for the environment. The added abstractions introduced by the architecture and framework thus allowed new environments with good user interfaces to be efficiently implemented.

Reuse of MViews has also indicated several areas which require more work to make software development environment construction easier and more general:

- MViewsDP's persistency management using Snart persistent objects proved to be much better than that of IspelM and MViewsER. A large amount of effort was expended in the development of IspelM and MViewsER just to define component data to be saved and restored, let alone defining incremental component saving and reloading. The Snart object persistency mechanism is a much more preferable approach with saving and reloading of component objects being almost transparent from MViewsDP (only calls to `sn_open_object_store, sn_close_object_store` and `sn_write_objects` needed to be added to the application class for MViewsDP). The Snart persistency mechanism, however, needs significant enhancement to support facilities such as version control and distributed, multi-user programming environments.

- MVSL and MVisual are useful for analysing environments and form a good basis for a design using the MViews architecture. MVSL should support component generalisation, however, and should allow subset components to explicitly state the base component attributes and relationships they view and

define automatic updating of these (where appropriate). It can be a tedious process replicating base component data in subset components and specifying update operations to maintain base and subset component consistency when the semantics of this process are usually well defined (updating a subset component attribute updates the base component attribute it views and vice-versa). Specifying how an update record is stored should also be more easily defined. MVisual should support calling MVSL operations as well as update operations (i.e. MVisual should be able to receive values back from MVSL by sending an event (update operation) to an MVSL component). Partial generation of MViews framework classes, parsers and unparsers from MVSL and MVisual descriptions would also greatly enhance environment development.

- Lazy update record processing proved to be very useful for composing update records into more abstract records (for example, translating `update_attribute` records for dialogue sub-components into `resize` and `move` records). It was also useful for determining whether to reconfigure a sub-icon to its parent's location (as the sub-icon has to determine whether it was sent a `shift_location` from itself and/or its parent). This lazy update record processing is quite low-level and only supported by the Snart framework. More powerful methods of specifying update record composition and lazy processing are required, particularly support for these at the modelling and architecture levels. Dependent component attribute recalculation could similarly do with more abstract specification and better framework support.

Based on the aspects of MViews which require more work, Chapter 10 discusses future research options which will help to improve MViews and the environments produced using MViews.

9.6. Summary

MViews has been reused to produce several novel environments and systems. All have a common underlying theme of canonical program representation based on object dependency graphs, multiple textual and graphical views of this program, and consistency management using update records. MViewsER provides graphical ER diagrams and textual RDS views. These views share some information and are kept consistent by update record unparsing and application and automatic component updating. MViewsDP provides a graphical dialogue view and textual dialogue specification and predicate views. All views share some information and are kept consistent via update records. MViewsDP also uses update records to maintain graphical icon positioning and composes abstract update records from fundamental update records to reduce the number of records stored, losing no information in the process. Cerno provides multi-object views of spied Snart objects and the views are kept consistent as

objects change. Classification of MViews component classes is used to achieve dynamic re-selection of object icon displays. Multiple, dependent subset components are used to view Snart lists and terms and keep these displays consistent with object data. Sorting animation and method tally graph views illustrate how MViews and Snart can be used to produce more abstract visualisations of object-oriented programs.

MViews has several other applications and MViews environments can be extended in various ways. ICAtect extends Snart, SPE and Cerno to support named facets for object attributes. Analysis and design diagrams for SPE could use update records to maintain analysis and design view consistency and allow programmers to specify more design detail. DFDs support analysis-level, abstract dataflow between classes and methods. Method DFDs provide a complementary visual programming technique for implementing methods. Tool abstraction allows systems to be decomposed into co-operating tools which are event-driven by tool and data structure changes. All of these systems reuse MViews' update records, object dependency graph representations and textual and graphical view abstractions.

Development of IspelM and the systems described in this chapter has indicated MViews provides a very useful set of building blocks for integrated software development environments. Of particular value are the novel MViews aspects of flexible object dependency graph representation, homogeneous use of update records for change propagation and documentation, and various abstractions for graphical and textual view and view components. Facilities that require further work include more flexible and concise MVSL and MVisual specifications, architecture and framework support for attribute recalculation and lazy update record processing, and more transparent and powerful component persistency management. Chapter 10 discusses some of the more important aspects of MViews that require further research and summarises the research contributions of MViews and MViews environments.

Chapter 10

Conclusions and Future Research

This chapter summarises the main contributions of this research to the field of software development environments. Using the discussions from the previous chapters, conclusions are drawn about the suitability of MViews for modelling and constructing ISDEs. The usefulness of IspelM, SPE and other systems developed using MViews (from Chapter 9) is also briefly discussed.

While MViews greatly enhanced the development of these environments, their development has indicated a need for a number of enhancements to MViews itself. These enhancements include: the need for more abstract attribute recalculation specification, lazy update record processing, and automatic support for update record composition; more abstract component persistency, similar to Snart object persistency; reusable version control and configuration management tools, and support for multi-user, collaborative software development; and partial generation of environments from MVSL and MVisual specifications, including unparser and parser generation. Important future research with SPE and IspelM includes: support for "typed" languages, where many inter-class relationships are automatically generated; reusing IspelM to produce environments for other object-oriented languages; and formal user evaluation of MViews environments, to determine how the provision of multiple textual and graphical views of software development with automatic consistency management assists, or hinders, software developers.

10.1. Research Contributions and Conclusions

10.1.1. Program and View Representation in MViews

MViews uses a novel object dependency graph mechanism for representing program structure and semantics as a canonical form in a shared data repository. Programs are represented as base program graphs which are comprised of elements (graph nodes) and relationships (graph edges). Subset views of this base program graph are constructed and are themselves program graphs comprised of subset element and relationship components. Subset view components are connected to base components with

relationship components. Program graphs are modified by a small set of graph editing operations.

This representation scheme is very general and flexible and can model both tree-based languages and graph-based languages. Language semantics can be modelled and stored as component attributes and subset views may represent and modify both structural and semantic information, as appropriate. Relationships between components determine inter-component dependencies and thus provide a structure for propagating component change without the need for a separate object dependency network.

Experience with MViews has indicated a need for an abstract component persistency mechanism. The MViews model and architectures can ignore the problem of program persistency if object persistency is supported by the implementation language for MViews systems. Providing architecture support for component persistency (via `save` and `load` methods) is a language-independent model but language-based object persistency has proved a much more abstract and easier to use approach.

10.1.2. Update Records for Change Propagation

MViews introduces the novel mechanism of update records for propagating a notification of the exact change to a program graph component. An update record is generated by a graph editing operation and propagated to all dependents of the updated component. These dependents are determined using the relationships the updated component participates in. Dependent components interpret update records they receive and may perform operations to reconcile their own state to that of the updated component they depend on. Update records provide a homogeneous solution to support different kinds of change propagation in ISDEs.

Update records are used by MViews environments to support a novel approach to graphical and textual view consistency. Update records may be unparsed in textual views to document a change to components represented in the textual view. An environment may also support automatic application of some update records to update the view's text forms, using incremental parsing and token substitution. This view consistency mechanism allows high-level graphical software representations and low-level textual program representations to be kept consistent automatically by an environment, no matter which view has been changed.

Update records are also used by MViews environments to support change propagation between base and subset components. This mechanism can support flexible, efficient attribute recalculation and lazy update record processing. As subset components are treated as dependents of their base components, update records provide a mechanism for keeping multiple views of shared program components consistent. The MViews

architecture supports a novel concept of subset/base relationships which can automatically translate between base and subset view attribute update records and corresponding operations. These subset/base relationships also support efficient view updating using lazy, demand-driven update record processing and their behaviour can be redefined to support very general viewing of base program components.

MViews environments support a novel "component self-documentation" scheme. Update records can be stored by a component to document changes the component has undergone. Using a similar principle, update records can be stored by subset views to implement a generic undo/redo facility for reversing and reapplying display and subset view editing operations. This mechanism is similar to database transactions, except a "transaction" is comprised of a sequence of update records, which are accessible to components and are used for several complementary purposes.

Update records allow a wide range of systems to be modelled including those utilising tool abstraction, multi-view editing, and general program dependency relationships. The most important, novel uses of update records have been supporting free-edited textual view consistency, dependent component state modification, and automatic component change documentation. The diverse uses of update records, and their good performance when implemented using Snart, suggests they are a very useful approach to handling general change propagation in ISDEs.

10.1.3. View Editing and Tool Interfacing

Subset views are rendered in either graphical or textual forms using display views. These display views also provide the editing tools for an environment with display view updates translated into subset view updates and vice-versa. MViews thus provides a novel tool integration mechanism via subset views whose rendering also supplies an editing tool for the environment. Display views are constructed from a common set of graphical user interface building blocks and may communicate (i.e. one view open another) via the subset and base views. This provides tight user interface integration with all editing tools having a common user interface. MViews supports the novel concept of interactively edited graphical display views and free-edited and parsed textual display views, with full view integration and consistency management via a canonical base program representation.

Subset and display views may also be used to integrate other tools into an environment. A subset view may simply provide access to the base view and translate data and editing operations to and from an external tool format. For example, a version control tool might be interfaced via a subset view and have a graphical user interface provided by a display view. Subset view components could also be used to relate external tool information (represented in a different format to MViews data) to base components. Subset views can

also propagate editing operations on base and subset components to external tools and translate external tool editing operations into subset and base component updates.

The provision of graphical and textual program views, with appropriate editing styles to the kind of view representation, has made MViews environments both easy and natural to use. Subset view consistency via unparsed update records or subset component modification in response to update records distinguishes the user interface of MViews environment views from other environments. While subset views have not yet been used for more general tool interfacing in MViews environments, they form a similar mechanism to that of ICAtect's external tool integration mechanism (Amor et al 91). This has been used successfully to integrate design tools with different data representations. Update record propagation between subset views and the base view provides a change propagation mechanism between tools of similar capability to that of FIELD environment selective broadcasting (Reiss 90a). Thus update records may prove useful for both data change propagation and editing operation propagation between external tools and MViews environments.

10.1.4. MVSL and MVisual

MVSL provides a specification language for defining the base and subset view states of an environment. MVSL component specifications can be augmented with extra graph editing operations built from a small set of fundamental operations. Update operations provide a mechanism for interpreting update records generated by components a component is dependent on.

MVisual provides a novel graphical specification notation for defining the user interface aspects of MViews environments. Display view and display component appearances are defined by example, as are dialogue and text form representations. A form of visual programming specifies the update record flow between MVisual graphical entities. This allows environment designers to define display view editing operations, the effect of display view editing operations on subset views, and the effect of subset view change on graphical entities.

Update records represent event flow between graphical entities and subset views and components and graphical entities and dialogue sub-component values can be specified in terms of subset view data defined in MVSL. MVSL and MVisual are currently assumed to communicate via update records passed between graphical MVisual entities and MVSL base and subset components. This loosely equates to the MViews architecture notion of display views and components being specialisations of subset views and components with communication by update records and method calls.

MVSL and MVisual are both useful for initial environment specification (as used for MViewsER and MViewsDP in Chapter 9) and environment documentation (as used for IspelM in Chapter 5 and Appendices D and E). MVSL proved useful for defining important component structures, operations and update responses. These specifications ignored considerations of exactly how attributes and relationships might be implemented and stored, implementation-level detail of component generation, propagation, storage and response to update records, and how base and subset view data is made persistent. MVisual allowed the appearance and basic functionality of display views for these environments to be defined for MVSL subset views. MVisual ignores implementation-level consideration of how subset and display views synchronise their updates and how display view editors are built. MVSL and MVisual specifications proved useful for the initial design of environments as they ignore detailed consideration of subset and display view communication, data storage, user interface functionality and efficiency, which have to be considered at the MViews architecture and framework levels.

Experience with these environments suggests partial generation of Snart framework classes from these specifications would enhance environment development. A crucial issue with this generation is how flexible the generated framework classes remain (i.e. how well they can be specialised to support activities not well expressed in MVSL and MVisual) and how consistency management can be employed between environment specification and implementation. MVSL should support more abstract language semantics specification, perhaps using a form of graph-based attribute grammars (Backlund et al 90, Hudson 91), and subset component specification in terms of base components. Improved communication between MVSL and MVisual would make both notations more expressive and useful.

10.1.5. MViews Architecture and Framework

Design and implementation of MViews environments uses an object-oriented architecture and framework of classes, rather than most approaches of generating environments from formal descriptions. The MViews approach is less abstract but more flexible than environment generation. It has resulted in environments which have user interfaces closely corresponding to those software developers prefer (such as interactively edited graphical software representations and free-edited and parsed textual representations).

The MViews architecture and framework provides a large range of useful component data and functionality. This allows environment developers to quickly design and implement environments based on MVSL and MVisual specifications using extra abstractions provided by the MViews architecture. This has resulted in very quick development of environments which have a lot of useful functionality. Other researchers have used reusable architectures to support programming environment and tool implementation (Nascimento and Dollimore 93, Newbury 88, Ratcliffe et al 92, Reiss 86). The MViews

architecture and framework, however, provide a novel set of reusable abstractions for both canonical program representation and integrated multiple textual and graphical view support. Use of these reusable build blocks has greatly reduced the time and effort taken to develop the environments described in this thesis than if MViews was not provided.

10.1.6. SPE and IspelM

SPE provides a novel, ISDE for analysing, designing, implementing and maintaining Snart software. SPE provides a concrete example of an environment based on the MViews model. Multiple textual and graphical views of software development are supported and these include abstract analysis and design class diagrams, textual analysis and design documentation views, textual program implementation views, and graphical debugging views using Cerno. Software construction views are kept consistent using update records and a canonical program representation.

SPE differs from comparable environments for object-oriented software construction by: its integration of graphical analysis and design views with textual program implementation views; its use of update records to maintain textual view consistency; and its automatic change documentation facility for program components. SPE also reuses an existing language compiler and run-time system, rather than re-specifying all static and dynamic language semantics within the environment. Use of SPE has indicated a need for more comprehensive analysis and design views of software development, version control facilities, and support for multi-user, collaborative software development. Determining the actual usefulness of SPE, via controlled user testing and evaluation, would provide a concrete demonstration of the worth of MViews environments.

IspelM reuses the MViews architecture and framework to provide a generic environment for object-oriented software development. IspelM demonstrates that MViews can be reused to model, design and implement such environments using abstract specification and object-oriented framework reuse. Reuse of the IspelM framework may indicate the need for further generalisation of the facilities provided by IspelM. IspelM framework reuse for Snart indicated language support for framework reuse may enhance the development of new environments.

10.1.7. Reuse of MViews

The environments developed using MViews in Chapter 9 help to demonstrate the flexibility of the MViews model and its wide range of applications. MViewsER provides a multiple view entity-relationship (ER) diagrammer with complementary textual relational database schema (RDS) views. MViewsER provides a novel mechanism for keeping these ER diagrams and RDS views bi-directionally consistent using MViews' update records. An advantage over comparable systems is that changes to ER diagrams not directly

affecting RDS views are unparsed for programmers to be made aware of. ER and RDS base component changes are automatically documented using update records providing a modification history similar to that of SPE.

MViewsDP provides a graphical dialogue painter view with textual views for specifying dialogue constraints, field defaults and return values. Use of MViews' update records to maintain graphical and textual view consistency, with automatic update record application to textual and graphical views, maintains full view consistency. This produces a novel environment where different views provide the most appropriate representation for different aspects of dialogue specification.

Method tally and sorting animation views illustrate that MViews can be used for various dynamic program visualisation applications. Cerno also illustrates this use of MViews but in addition demonstrates that the MViews architecture and framework can be usefully reused by other researchers. MViews and its derivatives are useful for constructing other software development environment tools and applications. These include more analysis and design views for SPE, dataflow method implementations for SPE, tool abstraction, and building model construction.

10.1.8. Snart

Snart provides a small contribution to object-oriented language development. Snart classifiers provide an imperative setting for dynamic classification, a language feature previously only supported by Kea (Hosking et al 90). Snart uses classifiers to support dynamic object class membership change, object feature spying for dynamic program visualisation, and object persistency.

Various improvements to Snart can be made including multiple object stores for object persistency, improved compilation optimisations, and better debugging facilities (such as those provided by Cerno). Adding typed object variables to Snart would allow SPE and IspelM to be expanded to cope with method arguments, automatic detection of inter-class relationships, and proper compile-time type checking. Strong typing also has interesting implications for the provision of imperative classification.

10.2. Future Research

Most of the previous chapters have contained brief discussions about possible future work on MViews environments. We confine the discussion in this section to the "hard" future research issues and leave discussion of cosmetic and performance improvements to the previous chapters.

10.2.1. Abstract, Flexible Component Persistency

Experience with MViews has indicated that component persistency should not be part of an environment's abstract specification or architecture. It has also shown that persistent objects may provide a mechanism of suitable abstraction and flexibility. Snart object persistency, however, needs to be enhanced to accommodate both flexible and efficient component object storage. MViewsDP uses Snart object persistency to store programs but only supports one object store being open at one time (due to the restriction imposed by Snart). This prevents the use of shared libraries and copying of components between programs which are not contained in a single object store database. In addition, Snart currently stores and loads objects individually to and from persistent storage. This can prove inefficient for components which are part of other components or usually require other components to be in-core (such as subset view components which are always reloaded with their owning subset view component by the MViews architecture).

Snart can be extended to handle multiple object stores in a similar way to how multiple object spaces are supported (used by SPE to separate SPE component objects and application objects). One or more object spaces could be defined which have a corresponding object store. Multiple object spaces used at one time require dereferencing of object identifiers according to which store they belong to. Copying of objects between stores could be supported by duplicating object data and copying it from one object store to another. Coarse-grained object storage could be supported by writing multiple objects to a single storage location and always reloading all these objects when one is accessed. The MViews architecture supports a form of this where multiple object `save_data` terms can be written to one text data file resource. This has indicated a much improved performance when loading closely inter-dependent objects over loading each object individually. Fig. 10.1. illustrates the format of a group of such object spaces and their object stores.
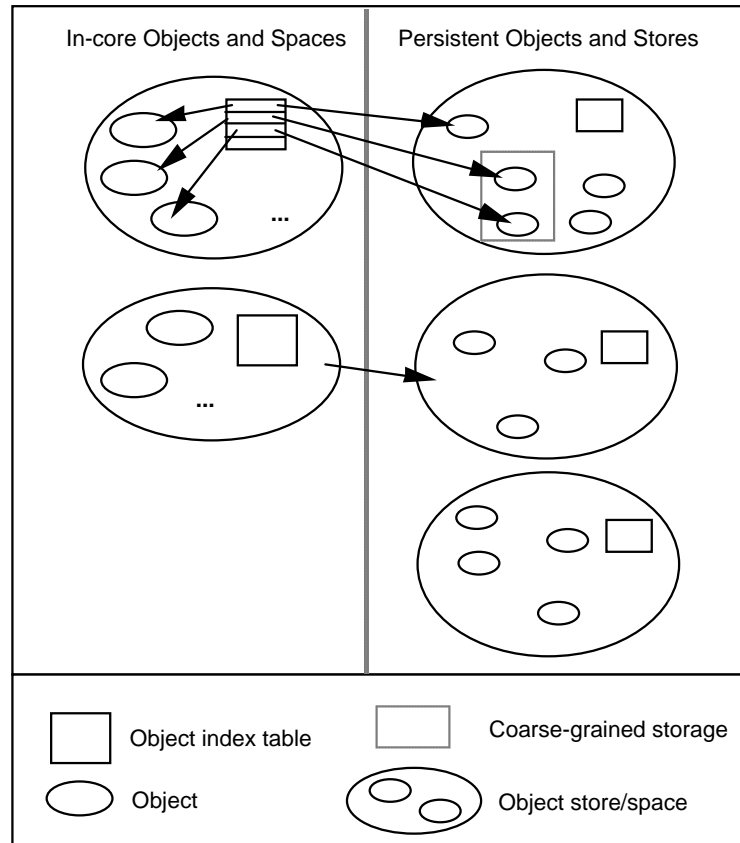
fig. 10.1. A group of shared Snart object stores and object spaces.

Some open research issues include how to abstractly specify the storage format of objects, how to update an object store when a class's interface is changed (i.e. object data is added or removed), and how to support a "deep copy" of objects from one store to another. For example, copying a class from one object store (say, a class library) to another (say, a program using the library class) may result in other classes used by the copied class needing to be copied and many component object references to be updated. The MViews architecture allows such component relinking to be hand-coded at present, but this is not a very abstract way of supporting such component duplication. The Snart object stores currently save their old class interfaces so objects with new class interfaces can be converted to their new forms. This is flexible but inefficient for large object databases.

10.2.2. Tool Integration Issues

MViews base views, subset views and update records provide a novel combination of view-based tool integration, canonical form program representation, and program dependency graph change propagation (see (Meyers 91) for a detailed discussion of these and other kinds of multi-view editing environment integration mechanisms). MViews supports the construction of editing tools by providing display views which render and manipulate subset view components. At present MViews does not provide any additional support for integrating external tools, as provided by FIELD (Reiss 90a) and Dora (Wang et al 92) environments.
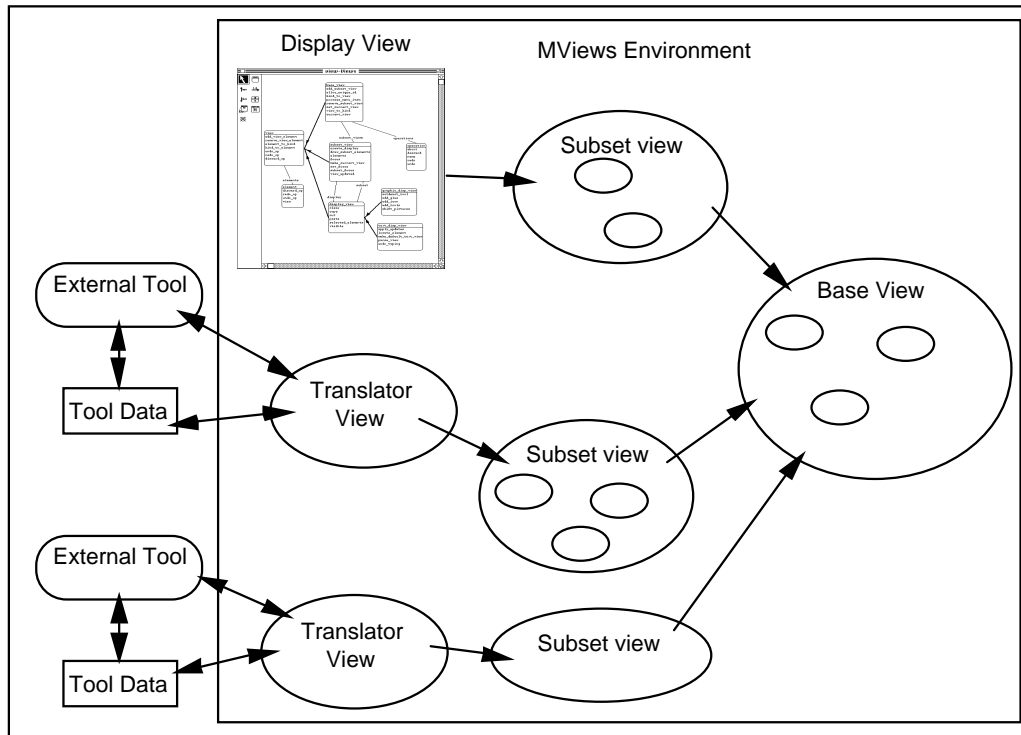
fig. 10.2. Examples of translator views for improved tool integration abstractions.

A possible approach to improving external tool integration would be to supply "translation" views for subset views, rather than display views. These translation views would have knowledge about the format of external tool data and may use subset view components to map external data entities to base program components. Editing operations from external tools may be converted into subset and base component operations. Update records generated by base and subset components translated into external tool editing operations or data modifications (if possible to do so). Translator views may need to parse data stored by external tools and unparse MViews component data in a similar manner to textual display views and components. A uniform approach to data parsing and unparsing may provide useful to facilitate both kinds of views. Translator tools may also require user interfaces (possibly display views) which provide a consistent user interface to external tools. Fig. 10.2. illustrates how translator views might interface between MViews and external tools.

10.2.3. Version Control and Configuration Management Tools

SPE currently does not support version control nor configuration management for software. As these facilities are common to most software development environments (Reiss 90a) reusable MViews tools to support these facilities would be useful. MViews environments can store update records to document the changes components have undergone. These stored records can form the basis of a flexible version control system.

Undoing and redoing the effect of a stored update records is straightforward. If stored update records are allowed to be undone and redone out of strict sequence, however,

MViews must check it is valid to carry out the operation. For example, undoing an attribute update if a component has been deleted does not make much sense unless the deletion is first undone. Two versions of a component could be merged by applying the update records of the second version to the component state described by the first version. Some update records may be invalid, however, and should be stored and reported to the programmer. Heuristics to automatically re-order update records so they can be applied could be supported (for example, undoing a deletion and then undoing an attribute update, as above). Added complications to this versioning process involve language semantics recalculation. Merging, reversing or reapplying version update records will require various language semantics to be rechecked to ensure a software system is still valid. A version control tool may need to be specialised for a given environment to support additional update records defined by components of the environment but not used by MViews.

A related issue to version control is configuration management where multiple versions of program components are held and a given program is configured from one version of each of its components. This has implications on the way MViews components are stored and a configuration management tool would need to ensure an object store has one version of each object at any one time. This may require multiple object stores for one group of program components or the configuration management tool may swap object versions to reconfigure the program version. Some open research issues include: how update records representing different versions are accessed by programmers from a current version of a program (probably via the configuration management tool); how to abstractly define configuration of components composed of other components (for example, a base class component has many feature components in SPE, and a different version of the class may have different versions of these features); the management of versions of subset and display views (different view versions require different base component versions and vice-versa); and storage of updated text form versions (where the "update record" contains the whole text form data).

10.2.4. Multi-user, Distributed Software Development

SPE currently supports only single user software development. Supporting multi-user software development could be achieved by providing each programmer a workspace made up of object stores for particular versions of a software system. Changes to these software versions could be exported to a shared program representation on-demand and versions from different programmers merged as appropriate. A programmer could also import updated versions from the shared representation and update their own versions. Fig. 10.3. illustrates such a multi-user, distributed software development environment for SPE.
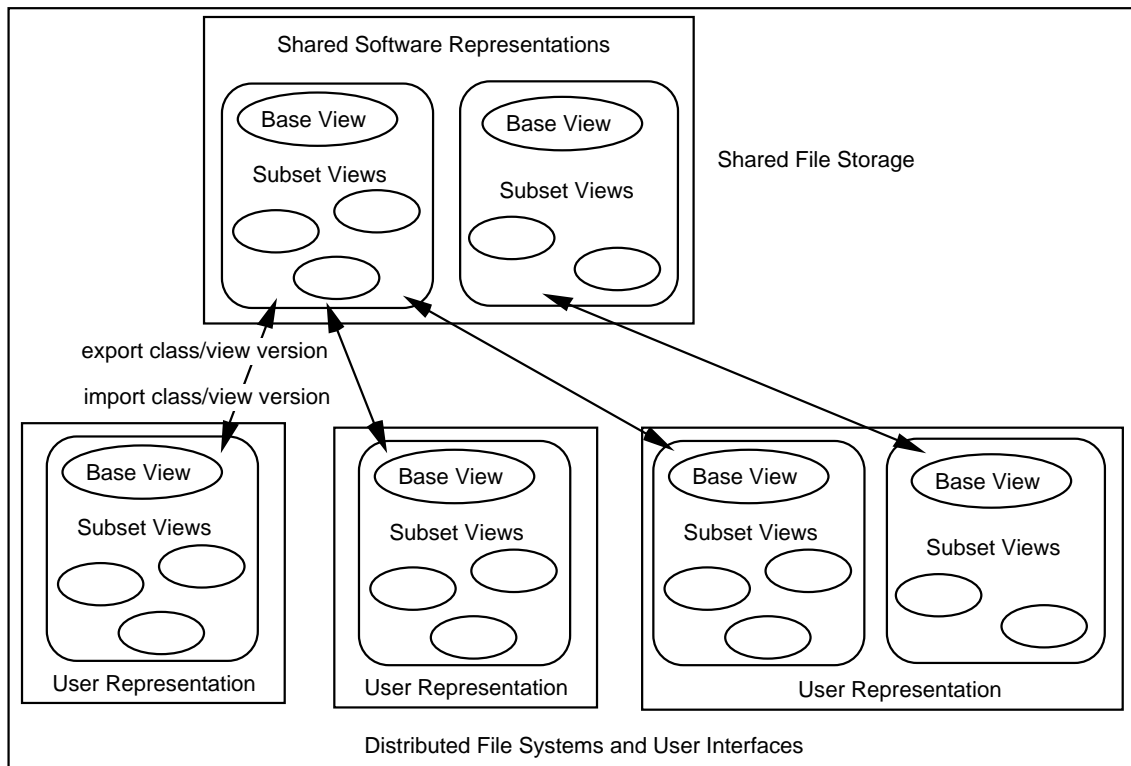
fig. 10.3. A multi-user, distributed software development environment.

Some research problems for supporting such an architecture include: abstract specification and efficient implementation of import and export tools; shared object store databases so multiple programmers can read update records and version information (object databases could be locked during version merging and an old object store database left readable); and support for collaborative software development. A disadvantage with this system is that programmers may spend a long time working on components which have been updated in new versions by other programmers. Update records could be broadcast to other environment invocations, similar to recalculated attribute values in Mercury (Kaiser et al 87), to inform other programmers of recent component updates they may need to be made aware of. Collaboration between programmers for software analysis and design may require views for these phases of development being kept consistent automatically by this update record broadcasting between environments.

10.2.5. Lazy Update Processing and Attribute Recalculation

MVSL does not currently provide any abstract mechanism for specifying language semantics using attribute grammars or any other formalism. MVSL also does not support the definition of update record composition and lazy update record processing. The Snart framework for MViews provides low-level support for these mechanisms based on update record storing and interpretation. Both specification and implementation of MViews-based environments would be easier if these facilities were more powerful.

An approach to providing these mechanisms at a higher level of abstraction is to extend MVSL so graph-based attribute grammars can be specified, at a similar level of abstraction to that of LOGGIE (Backlund et al 90). Implementation of these grammars could use the `get_attribute` method for component classes to recalculate attribute values (when required) and return them. `get_attribute` would need to be modified to register a component's dependencies on other component attributes. For example, fig. 10.4. shows an example attribute grammar based around MVSL components and a possible Snart framework implementation. The 4-argument form of `get_attribute` registers the given component and its attribute as a dependent of the recalculated attribute value (so the dependent attribute will be notified of a change to all attribute values it depends on). Further research is required to determine all the implications on attribute grammars of MViews component persistency, version control, and multi-user software development.

```
MVSL code:

base element class
   ...
   relationships
      attribute all_features : one-to-many all_feature is
         % calculate all_features from parents and
class.features relationship
         ...
end class

Snart framework code:

   base_class::get_attribute(Class,
      all_features,AllFeatures,DepComp,DepAttr) :-
      % calculate value
      ...

   Class@register_dependent(all_features,DepComp,DepAttr),
      ...

   base_feature::get_attribute(Feature,environment,Env) :-
      ...
      Class@get_attribute(
         all_features,AllFeatures,Feature,environment),
      % calculate value
```

fig. 10.4. Attribute grammars for MVSL and the Snart framework.

Lazy update record processing could be handled in a similar manner by specifying what update records are to be lazily processed and when their lazy processing should be scheduled. Update record composition could be supported by sending update records to a form of finite state automaton which composes a new update record from a sequence of basic update records generated by a component. One implication of this approach is how to determine when a composite update record shouldn't be generated. For example, if a

MViewsDP dialogue sub-component is resized in one editing operation it generates a composite update record instead of up to four update attribute records. If it is moved and then resized over subsequent operations, it should not generate a composite record.

10.2.6. Partial Generation from Abstract Specification

The MViews approach of abstract environment specification using MVSL and MVisual and then implementation using the Snart framework has proved very flexible. A disadvantage, however, is that much common information, particularly about program data representation and display view component appearance, must be replicated when implementing an environment.

Generating reusable Snart framework classes from an abstract environment specification may alleviate this problem. These generated classes would be reusable via specialisation and would thus overcome the traditional generated environment problem of lack of flexibility. Generating parsers and unparsers from grammars would assist the production of textual display views which are currently implemented using Prolog code.

10.2.7. IspelM and SPE Enhancements

Adding typed variables to Snart would allow experimentation with automatically generated inter-class relationships in SPE. Currently all inter-class relationships (such as aggregation, method calling and classification) are specified by programmers using graphical (and sometimes textual) display views. With a typed language many of these relationships can be inferred by parsing method implementations and class interfaces. This would generate a complete cross-reference database for classes, similar to that of FIELD environments (Reiss 90a and 90b). An implication of this automatic generation of relationships is how to efficiently update class relationships when a method or class is changed (so only modified relationships need be changed).

Specialising IspelM for other object-oriented languages may indicate the need for more object-oriented software development data and techniques to be stored at the IspelM level. For example, an environment for C++ software development could reuse SPE's analysis and design views with little or no change but requires different parsing, unparsing and compiler interface support. C++ classes have method argument types would should be represented (preferably at the IspelM level, as this is common to all strongly typed object-oriented languages).

The useful of SPE and other environments needs to be formally evaluated to determine just how such environments enhance software development. (Meyers 91 and Myers 90) note the difficulty in comparing new software development environments and visual programming techniques to conventional environments. The latter often have much more

well-developed tools while the former are often research projects lacking the fine-tuning and polishing of more traditional environments.

An approach to the evaluation of SPE might be to compare software development in SPE to Snart software development using the original LPA MacProlog-based Snart environment (as this simple environment has not been well-developed itself). One group of unfamiliar Snart programmers could be given SPE to use while another given the LPA environment. Development of a common program (such as the drawing program of Chapter 3) would provide feedback on how SPE provides better or worse support than the LPA environment. Developing more substantial programs in this manner would also be required and evaluation of programmers' subjective view of each environment may also prove very useful (as different programmers often like or dislike different aspects of software development environments (Glinert and Tanimoto 85)).

10.2.8. Imperative Classification and Framework Specialisation

Imperative classification in Snart has proved useful. Adding strong typing to Snart, as suggested in the previous section, has a number of interesting implications on the use of such imperative classifiers. For example, consider the example in fig. 10.5. from the drawing program in Chapter 3. The example on the left shows `drawing_window` method which converts a `rectangle` object into an `foval` object using the `shape` classifier. This works well as Snart uses run-time typing so the use of rectangle attributes before the classification and oval attributes after the classification is valid. The example on the right has a typing problem, as `Rect` is declared as a `rectangle`. After the imperative classification `Rect` now refers to an object of type `foval`!

```
drawing_window::rectangle_to_oval(Window,      drawing_window::rectangle_to_oval(
  Rect) :-                                        Window:drawing_window,
 Rect@height(Vertical),                           Rect:rectangle) :-
 Rect@width(Horizontal),                         Rect@height(Vertical),
 Rect@classify(shape,foval),                      Rect@width(Horizontal),
 Rect@v_radius:=Vertical,                         Rect@classify(shape,foval),
 Rect@h_radius:=Horizontal,                       % what does Rect now refer to?
 Rect@draw.                                       Rect@v_radius:=Vertical,
                                                  Rect@h_radius:=Horizontal,
                                                  Rect@draw.
```

fig. 10.5. An example of imperative classification in Snart and a typed version of Snart.

A solution to this might be to define the classify method as returning a new object reference of the appropriate type. This still does not stop `Rect`, or other object references to the classified rectangle, from assuming it is still an oval. Another solution might be to keep track of all references to classifiable objects and invalidate those that become incompatible with the object's new type.

Specialisation of IspelM to produce SPE indicated the need for language support for such framework specialisation. IspelM classes are not abstract classes and thus instances can be

created of them. One problem this causes in a specialisation of IspelM, such as SPE, is that objects must be created as instances of SPE classes, if an SPE class specialises an IspelM class. For example, base_class from IspelM is specialised to spe_base_class and all classes used by SPE must create instances of spe_base_class, not base_class. The MViews framework currently solves this problem by allowing environment implementers to override the base view method used to create new objects (kind_to_component) to support creation of the correct class instance. This is hardly an ideal solution as language-level support for framework specialisation would ensure the correct instance is always created.

10.3. Summary

MViews provides a novel model for ISDEs which support multiple textual and graphical views of information with consistency management. MViews provides a model based on object dependency graphs for representing program data and subset views of this program data. Subset views are rendered graphically or textually with graphical views interactively edited and textual views free-edited and parsed. The novel update record mechanism is used to maintain textual view consistency, propagate changes between components, support undo and redo of editing operations, and support component change documentation.

MVSL and MVisual support the specification of environments based on the MViews model. The MViews object-oriented architecture and Snart framework allow these environments to be implemented much more easily than without the MViews abstractions and building blocks. MViews has been reused to produce SPE, a novel ISDE for constructing Snart software. Other environments constructed by reusing MViews include an entity-relationship modeller with textual relational database views, a dialogue painter with textual constraint specification views, and various program visualisation views for SPE.

MViews can be extended in a number of ways to support the modelling and construction of ISDEs which provide flexible version control, multi-user, distributed software development, and partial environment generation from abstract specifications.

Glossary

Abstract class

A class which can not have any object instances. Usually used to factor out common data and behaviour and is a generalisation class for other classes.

Aggregation relationship

An aggregation relationship between two classes indicates that an instance of one class is composed of instances of the other class (i.e. a part-of relationship). For example, a drawing window object may be composed of zero or more button objects. Aggregation relationships are typically used for object-oriented analysis and are refined into client-supplier relationships.

Association relationship

An association relationship between two classes indicates one class makes use of the features of the other class in some way. For example, a figure class may be associated with a drawing window class, indicating the figure class uses the drawing window class in some manner. Association relationships are typically used for object-oriented analysis and are refined into client-supplier relationships.

Attribute grammars

Defining the static semantics of a software system in terms of attributes associated with the syntactic structures of the software system.

Attribute recalculation

The process of recalculating attribute values after an attribute value has changed or some syntactic modification has been made to a software system.

Base component

An element or relationship representing some base program information. Base components are part of the base view and may have zero or more subset view components linked to them.

Base view

A collection of program graphs which forms a canonical representation of an entire program.

Change propagation

The process of propagating an update (change) to one software system component to other components that may be affected by this change.

Class interface

The interface of a class is the set of all features defined by the class itself and inherited from all the class's generalisation classes.

Classification

> Dynamic classification of an object from membership of one class to membership of a descendant of this class. A classifier describes the set of valid descendant classes that can be used for the classification of the object and a class may define multiple classifiers. Classification is usually used to dynamically "specialise" an object' class membership using data supplied by a user at run-time.

Client-supplier relationship

> A client-supplier relationship between two classes indicates the client makes use of one or more features from the supplier. For example, a figure class may use the add picture method of a drawing window class and hence a client-supplier relationship from figure to drawing window exists.

Component

> Some view, element or relationship of an MViews system (may be at the base, subset or display level).

Concrete class

> A class which can have object instances (as opposed to an abstract class used for the purpose of factoring out common information).

Consistency management

> The process of keeping software system data consistent under change. For example, if a software system component is updated all other components dependent on the updated component's state must be updated in an appropriate way so the software system data is consistent. For example, if the interface to a class is changed, all classes using this interface should be re-checked to ensure they use the new interface correctly.

Display component

> An element or relationship that is part of a display view. Display components include icons (renderings of subset elements), glue (renderings of subset relationships) and textual display components (text forms which represent subset components).

Display view

> A graphical or textual rendering of a subset view. Display views also supply editor functionality for manipulating display components.

Features

> The features of a class are all the methods and attributes defined by the class, possibly including all inherited attributes and methods.

Generalisation relationship

Generalisation relationships between classes specify that a class is generalised to one or more other classes, and are typically used for object-oriented analysis. For example, a drawing window class is generalised to a window class and thus inherits all the functionality of this window class.

Graphical view

A graphical rendering of a view of a software system. Usually edited interactively or via structure-oriented editing (but may be free-edited using a drawing editor and parsed).

Inheritance relationship

Inheritance relationships between classes specify that one class inherits the data and behaviour of another class. Inheritance is typically used for object-oriented design and implementation and corresponds to generalisation. For example, a drawing window class inheriting from a window class inherits all the data and behaviour of this window class.

Object dependency graph

A dependency network between objects where each object has zero or more other objects dependent on its state. When the state of an object changes, its dependents must be notified of this change so they can update their own state appropriately to keep the system consistent.

Programming environment

An environment which assists programmers to implement and debug programs by providing tools for these tasks. Integrated programming environments provide an environment in which data and user interface integration is supported (i.e. a common data repository and common user interface is provided by the different tools).

Program visualisation

The process of viewing the static structure and/or dynamic behaviour of programs at a higher level of abstraction than program text and debugger trace. Usually utilises some form of graphical presentation.

Software development

The processes of analysing, designing, implementing, debugging and maintaining software systems. In a broader sense it also incorporates project management, collaborative software development, version control and configuration management. Integrated software development environments provide an environment in which data and user interface integration is supported (i.e. a common data repository and common user interface is provided by the different tools).

Software development environment

An environment which not only assists programmers to construct and debug programs, but includes support for other software development tasks, such as analysis, design and maintenance.

Software system

A super-set of a "program": software system data includes analysis and design information and may include debugging, maintenance, version control, and project management information.

Structure-oriented editing

Editing textual and graphical view components via a menu-driven and/or template style. Each syntactic program component is successively defined by expanding and filling in templates and this process does not permit syntactic errors to occur.

Subset component

A subset element or relationship which is a partial view of some base component. Updating a subset component is viewed as updating the base component in an appropriate way (for example, changing attribute X of the subset component is viewed as changing attribute X of the base component). Updating the base component may require the subset component to be updated in an appropriate way (for example, changing attribute Y of the base component is viewed as also changing attribute Y of the subset component). The subset component and its base component may not necessarily have the same attributes (for example, the base component may have additional attributes A and B while the subset component has attribute Z, and changing any of these attributes does not affect the other component).

Subset/base relationship

The relationship between a base component and a subset component. When the base changes, the subset component must be notified so it can reconcile its state to that of the base component. Similarly, when the subset component is updated it must translate the update on itself into an appropriate update on its base component. The MViews architecture provides a generic subset/base relationship which translates attribute updates between base and subset components (where they have attributes with the same name). This can be specialised to translate other updates appropriately.

Subset view

A partial view of the base view. Subset views are also made up of program graphs which are comprised of subset components.

Textual view

A textual rendering of a view of a software system. May be edited in either a free-edited or structure-oriented style.

Unparsed update record

An update record which has been unparsed into a readable form for inclusion in a textual display view to indicate a change in another view or in the update history browser dialogue. For example, the update record `update_attribute(Feature,feature_name,OldName,NewName)` might be unparsed into the form `% rename feature OldName to NewName`.

Update record

A description of the exact change applied to a software system component. For example, `update_attribute(Comp,Attribute,OldValue,NewValue)` describes an `update_attribute` operation applied to `Comp` renaming `Attribute` from `OldValue` to `NewValue`.

View

A view of a software system is a perspective on the system usually showing a subset of the entire system's state (i.e. a view contains a subset of all the elements and relationships between elements of the software system). Views are often rendered in various graphical and textual forms and two different views may represent the same software system data in the same or different ways.

View consistency

The process of keeping two independent views of the same software system data consistent under change. For example, if a representation of the data is modified in one view, this should be interpreted as the shared software system data being updated (in an appropriate way). Other views of this data should also be updated and re-rendered appropriately so all views correctly reflect the new state of the software system. MViews achieves view consistency by propagating update records between subset and base components which translate these update records into appropriate modifications on themselves.

Visual programming

The use of graphical views of program structure and/or behaviour to implement all or part of a program. Such views are typically edited using interactive or structure-oriented editing.

Appendix A

LPA MacProlog Facilities

This appendix gives a brief overview of the facilities provided by LPA MacProlog for constructing graphical user interfaces and "database" support. Snart programs can make direct use of these facilities by calling predefined LPA predicates. Alternatively, Snart classes can be defined which interface to these predicates to provide a class library for constructing user interfaces, similar to Interviews [Linton et al 88] and the THINK Class Library [Symantec 91].

A.1. LPA Graphics

LPA provides a Graphics Description Language (GDL) where graphical pictures are specified in a declarative way using Prolog terms [LPA 89b]. For example, fig. A.1. shows a GDL description of a class icon from SPE. The class icon description is a list of basic graphical pictures (i.e. a composite picture) including a filled, round rectangle (box), four text boxes containing the names of the class and features shown, and a line separating the class name and feature names. The location of each picture element is given in absolute co-ordinates. GDL descriptions can include modifiers like `blank(fillbox(...))` which indicates the round rectangle is filled with a blank (white) pattern. Other modifiers include shading of filled pictures with a variety of patterns, scaling and translation of pictures, and modification of the drawing pen size, colour and drawing mode.


```
figure

draw
hide
resize
```

Class Icon

```
[blank(fillbox(100, 100, 90, 60, 12, 12)),
    textbox('Courier', 9, 0, 110, 104, 12, 30, 0, figure),
    textbox('Courier', 9, 0, 130, 104, 12, 20, 0, draw),
    textbox('Courier', 9, 0, 145, 104, 12, 20, 0, hide),
    textbox('Courier', 9, 0, 160, 104, 12, 30 0, resize),
    line((126,100), (126,60))]
```
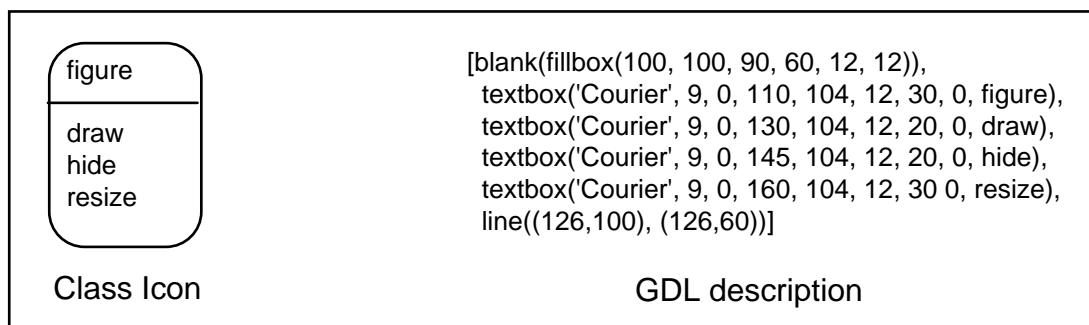
GDL description

fig. A.1.   A GDL description for a class icon.

GDL pictures are added to graphics windows, which are Macintosh windows supporting a drawing pane for rendering pictures and tool pane for manipulating pictures. Every graphics window has a list of pictures associated with it, and every picture in this list has a unique *name*,

a GDL *description*, a *local origin* (how much to shift the picture when rendering it), and a *selection flag* (selected pictures are highlighted with four squares around their frame).

LPA provides Prolog predicates to manipulate pictures in a variety of ways. For example:

- `add_pic(class_diagram1, class_icon1, [blank(fillbox(...)])` adds a new class icon picture to window `class_diagram1` identified by `class_icon1`;

- `del_pic(Class_diagram1,class_icon1)` removes `class_icon1` from `class_diagram1`;

- `get_pic(Class_diagram1,class_icon1,Description)` binds `Description` to the GDL picture description for `class_icon1`;

- `sel_pics(Class_diagram1,[class_icon1,...])` selects the list of given pictures;

- and `shift_pics(Class_diagram1,[class_icon1,...],(YDelta,XDelta))` moves the named pictures by `YDelta` and `XDelta`.

Picture elements are drawn in order (i.e. the first element drawn first, the second over the top of the first, and so on) and a window's picture list is drawn in reverse order (i.e. the last picture name being drawn first, the second to last next, and so on). LPA automatically redraws pictures if they are affected by a change to other pictures in a window. This relieves the need for programmers to implement their own window refreshing algorithms and greatly simplifies such tasks as shifting pictures and modifying picture descriptions.

Graphics windows are composed of a drawing pane, a tool pane and an optional viewing pane. Graphical *editing tools* are associated with a window and any editing operations applied to the drawing window (clicking on pictures, dragging pictures and so on) are sent to a predicate defined by the current editing tool. LPA provides predicates to implement rubber-banding, marqui selection of pictures and cut-and-paste operations. Tool building predicates are provided for editing text selections, dragging selected pictures and processing mouse clicks. Advanced support for incremental picture redrawing and programmer-managed validation and invalidation of drawing regions is also provided.

A.2. LPA Menus and Dialogs

LPA provides a comprehensive range of declarative menu and dialog specification predicates. Fig. A.2. shows a menu definition for IspelM. Selecting a menu item results in a predicate call of the form `MenuName(MenuItem)`. Menu extensions such as checked items, menu styles, fonts and picture items are also supported.

Dialog specification is one of the most useful aspects of LPA's graphical user interface support. Dialogs are defined in a similar manner to menus with a single predicate call. Fig. A.3. shows a dialog definition from IspelM.

Default values for dialogs are supplied by the specification of a dialog (and can be bound Prolog variables passed to the predicate defining the dialog). Returned values are passed back

in variables that were unbound on creation of the dialog. Both modal and non-modal dialogs are supported with check boxes, radios, text fields, edit fields, pictures and menus being supplied by LPA. Validation of input can be performed by providing a predicate to be called as part of the dialog specification. Using Prolog to implement declarative input checking predicates in conjunction with declarative dialog specification worked very well in the implementation of MViews and IspelM.

```
                                    % The Views menu.
                                    %
                                    mv_create_ViewsMenu :-
                                     kill_menu('Views'),
                                      install_menu('Views',
                                        ['Create Element View',
                                         'Kill View',
                                         'Rename View',
                                         'Make Current View',
                                         '(-',
                                         'Select Element View',
                                         'Select A View',
                                         '(-',
                                         'Make View Focus',
                                         'Make Default Text View']).

                                    'Views'(Item) :-
                                     mv select menu item('Views',Item).
```

Views Menu	Views Menu Specification

fig. A.2.   A menu description from IspelM.

```
% Get feature details.
%
is_feature_details(OldName,NewName,OldKind,NewKind,
  OldType,NewType,Show,Action) :-
 centred(Top,Left,200,290),
 is_convert_feature_kind(OldKind,
   OAttribute,OMethod,OAbstract,OInherited),
 !, mdialog(Top,Left,200,290,
  [button(110,150,20,60,'Change'),
   button(170,220,20,60,'Cancel'),
   button(140,150,20,60,'Remap'),
   button(110,220,20,60,'Hide'),
   button(140,220,20,60,'Remove'),
     text(10,10,16,220,'Feature name:'),
     edit(30,10,16,220,OldName,gread(NewName)),
     text(50,10,16,220,'Feature type:'),
     edit(70,10,16,220,OldType,gread(NType)),
     radio(110,10,16,90,'Attribute',OAttribute,NAttribute),
     radio(130,10,16,90,'Method',OMethod,NMethod),
     radio(150,10,16,90,'Deferred',OAbstract,NAbstract),
     radio(170,10,16,90,'Inherited',OInherited,NInherited),
     check(90,120,16,110,'Show Feature',on,NShow),
     text(90,10,16,100,'Feature kind:')],
  Btn,is_check_feature_details(NewName,NType,NAttribute,
    NMethod,NAbstract,NInherited,NewKind,NShow,Show,Action))
 is_blank_or_atom(NType,NewType).
```

Feature Details Dialog	Dialog Specification

fig. A.3.   A dialog description from IspelM.

LPA supports text windows that provide an editing pane where users perform conventional text editing operations (typing text, cut, copy and paste, and text selection and deletion).

A.3. LPA File and Resource Management

LPA provides access to the Macintosh file system including resource file management. Files are treated as file name/volume id pairs. Resources are identified by numeric or atomic values and in the current version of LPA store Prolog atom values (i.e. up to 255 text characters). Prolog predicates can be saved in either text or code forms and reloaded incrementally or as an entire "image".

These facilities can be used to provide a rudimentary database in which arbitrary Prolog data items can be stored. An arbitrary Prolog data item can be written to a text window and the text window contents written to a file or resource (in the latter case, using several resources if the window's text is greater than 255 bytes, as described in Chapter 7). Given the "address" of the data item's text in a file (either as a file position or resource ID) the data can be converted back into a Prolog term by performing a read at the file position, or reloading the text into a window from resources and reading from the window.

Appendix B

The Snart Language

This appendix gives a more complete description of Snart than Chapter 3. The language syntax and run-time object manipulation predicates and methods are described, together with the extended LPA MacProlog environment for Snart. A description of the compiler and run-time system implementation is given. A comparison of language features and general philosophies is presented between Snart and other object-oriented Prolog systems. A brief description of a version of Snart ported to Quintus Prolog is supplied together with some proposed extensions to the language itself. Snart currently runs under LPA MacProlog (LPA) on the Macintosh (version 4.5). For more information about LPA, see (LPA 92).

B.1. Syntax

Snart programs are composed of three basic parts:
- *class definitions* which define the name and kind (abstract or normal) of a class, the parents and renamed features for the class, and the features for the class (attributes and their types, methods, deferred methods and classifiers)
- *method predicates* which implement methods for a class
- *prolog predicates* which interact with Snart method predicates (by being called by method predicates and/or accessing Snart objects)

Snart class definitions, method predicates and Prolog predicates can be in the same LPA program window, or in different program windows.

B.1.1. Class Definitions

Fig. B.1. shows the syntax for a Snart class definition. Abstract classes can not have instances created of them. Parent classes, if specified, must exist and must not inherit from the class being defined. Renamed features must also exist (i.e. be inherited from the parent class) and must not be renamed to names of features either being defined in this class or being renamed from other parent classes. Features are redefined in the new class by just repeating the name used in the class's parent.

Attribute types are not currently used except when defaulting values for an attribute (see below). However, they should be valid attribute types as defined above. Classes in the class name list of a classifier must exist, be sub-classes of the class being defined, and there must be no repetition of a class or any of its descendants in the class name list (i.e. all classes in the class name list must be disjoint under inheritance from the class being defined).

```
ClassDefinition ::=                        FeatureList ::=
  ClassKind '(' ClassName ','                /* empty */
    parents '(['ParentList '])' ','          | FeatureDefinition
    features '([' FeatureList '])' ')';      | FeatureDefinition,
                                                 FeatureList;
ClassKind ::= abstract_class | class;
                                           FeatureDefinition ::=
ClassName ::= atom;                           Attribute
                                            | Method
ParentList ::=                              | DeferredMethod
  /* empty */                               | Classifier;
| Parent
| Parent ',' ParentList;                   Attribute    ::=    FeatureName    ':'
                                           AttributeType;
Parent ::=
  ClassName                                AttributeType ::= atom | term | boolean |
| ClassName '([' RenameList '])';             integer | string |
                                              list '(' AttributeType ')' | ClassName;
RenameList ::=
  rename '(' FeatureName ',' FeatureName   Method ::= FeatureName;
')'
|  rename '(' FeatureName ',' FeatureName  DeferredMethod ::=
')' ','                                       FeatureName '(' deferred ')';
  RenameList;
                                           Classifier ::=
FeatureName ::= atom;                          FeatureName ':' '[' ClassNameList ']';

                                           ClassNameList ::=
                                              ClassName
                                           | ClassName ',' ClassNameList;
```

fig. B.1.   Syntax for Snart class definitions.

B.1.2. Method Predicates

Fig. B.2. shows the basic method predicate syntax for Snart.

```
MethodPredicateDefinition ::=              MethodPredicateDefinition ::=
  MethodPredicate.                             PrologPredicate;
| MethodPredicate :-
  MethodPredicateDefinition;               ObjectVariable ::= PrologVariable;

MethodPredicate ::=                        ArgumentList ::=
 ClassName '::' FeatureName(                  PrologTerm
  ObjectVariable ',' ArgumentList);        | PrologTerm ','
                                              ArgumentList;
```

fig. B.2. Syntax for Snart method predicates.

Multiple clauses for method predicates can be defined, with the appropriate clause being executed in the same manner as for standard Prolog predicate clauses. The ClassName for a method predicate must be a class name with a class definition and the FeatureName used must be a method of this class. The first argument of a method predicate is always a variable bound to the object ID of the object the method is being executed for (i.e. the

object sent the FeatureName message). Further arguments are bound to the remaining arguments in the feature call.

B.1.3. Prolog Predicates

Prolog predicates may be defined use the conventional LPA Prolog predicate syntax. These may be positioned before or after method predicates and class definitions they are associated with, or defined in different program windows.

B.2. Objects

Objects are created using the `create` method common to all classes. Feature calls are made using the `@` operator, and attribute assignment using the `:=` operator (fig. B.3.).

```
Object@create(ClassName,ArgumentList)        Object@AttributeName:=AttributeValue

Object@FeatureName(ArgumentList)             AttributeValue : PrologTerm;
```

fig. B.3. Syntax of object creation, feature calling and attribute assignment.

The create method call can have only a ClassName argument, in which case the new object is just created and any create method predicate defined for it is not called. A feature call may be a fetch of an attribute value (in which case it has one argument being a variable or value which is unified with the attribute value of the object). If no such attribute value has been assigned for the object, the call fails. If no such attribute exists for the object's class, execution of the program is aborted with an error message displayed.

If a feature call is a method call, the appropriate method predicate for the object is called with the first argument being the object ID, the rest being the arguments given to the feature call. If the method predicate call succeeds then the feature call succeeds, otherwise the feature call fails. If the method doesn't exist for the object's class, execution is aborted.

Attribute assignment stores a value associated with the object which can be accessed by a feature call of the form `Object@AttributeName(Value)`. Attribute assignment always succeeds unless the attribute doesn't exist for the object's class, in which case execution is aborted.

Further method calls defined for all objects are described in table B.1.

Method	Description
`Object@dispose`	Disposes of Object
`Object@copy(-NewObject)`	Duplicates Object and returns NewObject as duplicated Object ID

`Object@member(+ClassName)`	Succeeds if Object is a descendant of ClassName
`Object@default(+Attribute,?Value)`	Returns value of Attribute for Object, or default value for type if no value exists
`Object@default(+Attribute,` `+Default,?Value)`	Same as default/2, but returns Default if no value exists for Attribute for Object
`Object@is object`	Succeeds if Object a valid Snart object
`Object@object attribute(+Attribute)`	Succeeds if Attribute a valid attribute for Object
`Object@object attribute(+Attribute,` `?Type)`	Returns type of Attribute for Object, fails if Attribute not a valid attribute for Object
`Object@print`	Prints out attributes and class for Object
`Object@classify(Classifier,` `ClassName)`	Classifies Object to ClassName using Classifier classification attribute
`Object@AttributeName?=Value`	Backtrackable attribute assignment, resets old value on failure

table B.1. Method calls defined for all Snart objects.

The object manipulation methods described in table B.1. can also be invoked on an object as a predicate. Table B.2. shows the correspondence of Methods to predicates. Predicate invocation is more efficient, as it by-passes the Snart method despatcher.

Method Call	Predicate Call
`Object@dispose`	`delete_object(+Object)`
`Object@class(?ClassName)`	returns the class an object belongs to
`Object@copy(-NewObject)`	`copy_object(+Object,-NewObject)`
`Object@member(+ClassName)`	`member_class(+Object,+ClassName)`
`Object@default(+Attribute,?Value)`	`default_value(+Object,+Attribute,?Value)`

`Object@default(+Attribute,`
  `+Default,?Value)`
	`default_value(+Object,+Attribute,`
	  `+Default,?Value)`

`-`	`is_object(+Object)`
`Object@object_attribute(+Attribute)`	`object_attribute(+Object,+Attribute)`

`Object@object_attribute(+Attribute,`
  `?Type)`
	`object_attribute(+Object,+Attribute,`
	  `?Type)`

`Object@print`	`print_object(+Object)`

`Object@classify(Classifier,`
  `ClassName)`
	`classify_object(+Object,+Classifier,`
	  `+ClassName)`

`-`	`is_class(+ClassName)`
`Object@AttributeName?=Value`	`-`

table B.2. Method calls and their equivalent predicate calls.

B.3. Environment

To support Snart programming, the LPA environment has been extended. Extra menu items are added to provide location facilities for classes and method predicates, printing of class and object data, compilation and optimization of classes, and deletion of objects.

B.3.1. Search Menu

Fig. B.4. (a) shows the Search menu from Snart. The LPA menu items are Find... through to Call graph. These are documented in the LPA Environment manual (LPA, 89b).

```
┌─────────────────────────┐  ┌─────────────────────────────┐
│ █Search█                │  │ █Eval█                      │
│ Find...              ⌘W │  │ Query...                 ⌘Q │
│ Find next            ⌘F │  │ Run last query           ⌘U │
│ ........................ │  │ Enter                    ⌘T │
│ Change...            ⌘G │  │ ........................... │
│ Replace & find next  ⌘R │  │ Compile                  ⌘K │
│ ........................ │  │ Compile selected            │
│ Find selected        ⌘E │  │ ........................... │
│ Find definition...   ⌘D │  │ Set spy points...           │
│ ........................ │  │ Clear all spy points        │
│ Get information...   ⌘I │  │ ✓Debug                      │
│ Call graph              │  │ ✓Leash                      │
│ ........................ │  │ ........................... │
│ Find Class...           │  │ Compile Classes          ⌘H │
│ Find Selected Class  ⌘L │  │ Compile All Classes         │
│ Find Named Class...  ⌘J │  │ Compile Selected...         │
│ Find Method Defn...  ⌘N │  │ Compile Named Classes... ⌘P │
│ ........................ │  │ ........................... │
│ Find Object...       ⌘O │  │ Spy Method...               │
│ Find Named Object...    │  │ No Spy Method...            │
│ ........................ │  │ Clear Method Spys           │
│ Print Class...          │  │ ........................... │
│ Print Selected Class    │  │ Delete Class...             │
│ Print Named Class...    │  │ Delete Named Classes...     │
└─────────────────────────┘  │ Delete Objects           ⌘M │
                             └─────────────────────────────┘
```

figs. B.4 (a) and (b).   The Snart Search and Eval Menus.

Find Class...

Find Selected Class

Find Named Class...

> Find Class opens a dialog with all currently defined classes listed in a scrolling menu. After selecting a class, the class definition is searched for in the LPA program windows, and the class definition highlighted and its window brought to the front. Find Selected Class locates and highlights the class name given by the currently selected item of text in a window. Find Named Class asks for the name of a class and highlights it.

Find Method Defn...

> A dialog box with edit fields for the class and method names is opened. The user keys the names and the method predicate for the class is located and highlighted. Alternatively, the user may request menu dialogs of the currently defined classes and the methods for a class name, and choose from these.

Find Object...

> A dialog box with edit fields for an object ID, class name, and attribute values is opened. After entering data into one or more fields, Snart searches all currently defined objects for one or more that matches the description. If one is found, a listing of the object's class and attribute values is printed in a display window. If more than one is found, their object ID's are printed.

Find Named Object...

> An object ID is requested and the object's attribute values and class printed in a display window.

Print Class...

Print Selected Class

Print Named Class...

    Print Class requests a class from a menu dialog of all currently defined classes. The class parents, attributes and methods are printed in a display window. Print Selected Class prints details for the class name given by the currently selected item of text in a window. Print Named Class asks for the name of a class and prints its details.

B.3.2. Eval Menu

Fig. B.4. (b) shows the Eval menu from Snart. The LPA menu items Query... through to Leash are documented in the LPA Environment Manual (LPA, 89b).

Compile Classes

Compile All Classes

Compile Selected...

Compile Named Classes...

    Compile Classes recompiles the class definitions for all classes in modified program windows. Their definitions are only recompiled if they have changed from their previously compiled definition. Sub-classes are recompiled if their parents have altered. Compile All Classes forces the recompilation of all Snart classes whether they have been modified or not. Compile Selected compiles all classes selected from a menu list. Compile Named Classes recompiles all classes entered by the user.

Spy Method...

No Spy Method...

Clear Method Spys

    Spy Method requests class and method names, and sets a debugger spy point on the selected method. No Spy Method... clears the spy point on the selected class and method. Clear Method Spys clears all spy points on Snart method predicates.

Delete Class...

Delete Named Classes...

Delete Objects

    Delete Class deletes the compiled definition of a selected class. Delete Named Classes deletes the definitions of classes typed in by the user. Delete Objects destroys all currently defined Snart objects.

B.3.3. Defining and Compiling Classes and Method Predicates

Snart class definitions and method predicates are defined in the same windows as LPA Prolog code. After entering or modifying Snart code, the Snart classes are recompiled (using the Compile Classes menu item) before invoking a Prolog predicate that uses the classes. Class and method definitions may be modified but left uncompiled during debugging.

Class definitions are converted into an internal form when a window is recompiled and their method despatch tables are regenerated when the compiler is invoked. Method predicates are basically ordinary Prolog predicates called on a message invocation, thus methods use standard Prolog code and Snart feature calls.

Snart objects can be either deleted after their classes have been modified, and the program re-executed to re-build them, or left intact and used with modified method predicate and class definitions. If an old object is used (e.g. attribute accessed) with a new class and an incompatibility occurs, Snart notifies the user of an error.

Errors in Snart (e.g. accessing an invalid attribute, calling a nonexistent method) are all run-time errors detected by the Snart predicates. These cause termination of the current Prolog process with appropriate debugging information. The only compile-time errors detected are non-existent classes or duplicate feature names for a class.

B.3.4. Debugging Snart Programs

The LPA debugger is used to debug Snart method predicates. Snart objects can be printed for debugging using the Search menu, and spy points set and cleared on methods using the Eval menu. When tracing a method call, the LPA debugger calls the Snart method despatcher to determine the attribute fetch or method predicate call to make. Programmers can then decide whether to **Leap** through the call or **Creep** through the method predicate code.

B.3.5. Saving Snart Programs

Snart classes and method predicates are saved along with normal Prolog code they are defined with. On reloading a Snart program, the classes must be recompiled to restore their definitions. Object code can be saved, and on loading the Snart classes do not have to be compiled (the Snart compiler does not have to be loaded, only the Snart object predicate definitions). Currently, no support is provided to make Snart objects persistent. Object data can be saved as Prolog terms, and new objects created when the old data is reloaded.

B.4. Compiler

Snart class definitions and method predicates are pre-compiled before invoking a Prolog predicate that makes use of Snart. Thus we have a compilation phase similar to that of object-oriented languages like Eiffel (Meyer, 88) and C++ (Winblad et al, 90).

B.4.1. Class and Method Predicate Storage

When LPA compiles program windows, class definitions and method predicates are read with conventional LPA Prolog code, and are predicates of the form:

```
ClassDefinition = ClassKind(ClassName,parents(Parents),features(Features))
MethodPredicate = ClassName::MethodName(ObjectID,Arg₁,...,Argₙ) :-
      MethodImplementation.
```

Class definitions must be converted into a predicate form with a unique predicate name, rather than using "class" or "abstract_class", as LPA Prolog does not allow predicate definitions to be defined over different windows, or to be separated by other predicates. Method predicates use an infix operator ':::' to link their class and method names, which must be converted into a single atomic form.

For optimallity, the Snart compiler only recompiles a class if its definition has been recompiled by LPA since the last invocation of the Snart compiler. Snart keeps an integer property called `sn_last_compile` to remember the last compilation "time"of a class definition. This is incremented each time a compilation is performed, and only class definitions with numbers higher than the last compile are checked. Thus Snart must also change class definition predicates to record the last time they were compiled by LPA.

Snart uses `term_expansion/2` to convert class definitions and method predicates into an internal form. Class definitions are prefixed by their name (with a 'c_' appended to ensure they are unique from any other Prolog predicates), and have the last compilation time number added to their definition. Method predicates have their class and method names joined to form an atomic predicate name that can be directly called by the Snart method despatcher. Fig.s B.6.. and B.7. illustrate this translation process using the drawing program from Chapter 3 as an example.

```
abstract_class(figure,                      class(rectangle,
  parents([figure]),                          parents([
  features([                                    closed_figure([
    visible(boolean),                             rename(create,fig_create)
    create,                                     ])
    draw,                                     ]),
    resize(deferred)                          features([
])).                                            width:integer,height:integer,
                                                create,draw,resize
figure::create(FigureID) :-                   ])).
      FigureID@visible := false.
                                            rectangle::draw(RectangleID) :-
                                              RectangleID@width(Width),
                                              RectangleID@height(Height),
                                              draw_figure(rectangle,Width,Height).
```

fig. B.6.   Rectangle class and method predicate definitions.

Snart must also have a mechanism for quickly finding all defined classes. In addition to translating the class definition predicate into an internal form, Snart records the class name as a property of the form:

```
<ClassName,sn_class_defn,PredicateName>
```

where:

```
PredicateName = concat('c_',ClassName)
```

```
c_figure(10,[],abstract,                  'c_rectangle(10,[closed_figure([
  [visible(figure,attribute,boolean),        rename(create,fig_create)])],
   create(figure,method,'figure::create'),   [width(rectangle,attribute,integer),
   draw(figure,method,'figure::draw'),       height(rectangle,attribute,integer),
   resize(figure,deferred,'')]).             create(rectangle,method,
                                               'rectangle::create'),
'figure::create'(FigureID) :-
  FigureID@visible := false.               draw(rectangle,method,'rectangle::draw'),
                                            resize(rectangle,method,
rectangle::draw'(RectangleID) :-             'rectabgle::resize')]).
  RectangleID@width(Width),
  RectangleID@height(Height),
  draw_figure(rectangle,Width,Height).
```

fig. B.7. Using `term_expansion` to convert rectangle class into internal form.

B.4.2. The Compilation Process

When a programmer has modified some class definitions or method predicates and wants them recompiled, `sn_compile_classes/0` is called. Snart firstly finds all classes using the `sn_class_defn` property. It then determines if the class's definition has been modified (using the `sn_last_compile` property, which is incremented by one). For each modified class definition, Snart recompiles the class.

To recompile a class, Snart processes the class's features and parents lists. Each parent class is recompiled if it has been updated since the last compilation. Features are inherited from each parent, and are renamed according to the rename list associated with each parent. Any duplicate features caused by multiple, repeated inheritance are removed. The parents and class features lists are then merged with any features redefined in the class replacing the parent features of the same name. The merged list is then checked for any duplicates. Fig B.8. illustrates this process.

```
Compiling rectangle...
% class data
Parents = [closed_figure([rename(create,fig_create)])]
Features =
  [width(rectangle,attribute,integer),
   height(rectangle,attribute,integer),
   create(rectangle,method,'rectangle::create'),
   draw(rectangle,method,'rectangle::draw'),
   resize(rectangle,method,'rectangle::resize')]
% Compile closed_figure if necessary...
ParentFeatures=
  [visible(figure,attribute,boolean),
   fig_create(figure,method,'figure::create'),
   draw(figure,method,'figure::draw'),
   resize(figure,deferred,'')]
% Merge ParentFeatures and Features reporting duplicates
(if any)
ObjectFeatures =
  [width(rectangle,attribute,integer),
   height(rectangle,attribute,integer),
   create(rectangle,method,'rectangle::create'),
   draw(rectangle,method,'rectangle::draw'),
   resize(rectangle,method,'rectabgle::resize'),
   fig_create(figure,method,'figure::create'),
   visible(figure,attribute,boolean)]
ObjectParents = [closed_figure]
```

fig. B.8. Inheriting features and attaching method predicate names for a class.

The Snart compiler now determines if this new definition has altered since the last time it compiled the class. The compiled class format is looked up and compared to the new format. If either of the features or parent lists has changed, or the class's kind has changed, the new class definition is stored and the method dispatch table for the class regenerated. If the class has changed, any children of the class are recompiled (to ensure their definitions are kept consistent). Snart stores the children of a class using a list property `sn_children`, which is updated each time a child class is compiled.

The compilation process is complete once all updated classes have been recompiled (and the transitive closure of their children have been too, if necessary). Note if a class definition has not yet been compiled (i.e. it has just been added by a programmer), it is always compiled. Its `sn_children` property is set to [], and as any children of the class are found, their names are added to this list.

B.4.3. Compiled Class Format

Snart stores compiled class definitions as a property of the form:

```
<ClassName,sn_class_data,ClassKind(ObjectParents)>
```

In addition to this predicate, Snart sets properties for each method and feature for the class, of the form:

```
<ClassName,MethodName,MethodPredicate>
<ClassName,AttributeName,attribute(AttributeType)>
```

These are used by the method despatcher for quick look-up of a class's method predicates and attribute and classifier types. Deferred methods are not added to the despatch table and abstract classes have no method or attribute despatch tables generated (as no instances of abstract classes can be created). The full despatch table is generated for a class (including all its inherited features) for maximum speed. Alternatively, only features defined for a class (including its renames) can be generated and features looked up at run-time by searching a class's ancestors.

B.5. Run-time System

B.5.1. Object Creation, Attributes and Destruction

Snart objects are identified by an integer atom (which is used as their object ID). On object creation, a property of the form:

```
<ObjectID,sn_object,ClassName>
```

is set, which identifies the class an object belongs to. All Snart object predicates use this property to determine whether an object is valid, and what class it belongs to.

When an object attribute is set by:

```
ObjectID@AttributeName:=Value
```

Snart checks the object is valid (by accessing its `sn_object` property) and checks the attribute name is valid (by calling `ClassName(AttributeName,attribute,Type)`). Then Snart sets a property of the form:

```
<ObjectID,AttributeName,Value>
```

On object attribute access of the form:

```
ObjectID@AttributeName(Value)
```

Snart simply looks up a property of the form:

```
<ObjectID,AttributeName,Value>
```

and returns Value bound to the property value. If the attribute value property is not found, Snart checks the attribute name's validity for the object's class. If valid, the object lookup fails, otherwise an exception is raised and the current Prolog process aborts.

On object deletion, Snart checks the object is valid, and then removes all attribute properties associated with the object (including `sn_object`).

B.5.2. Method Calls

On a method call of the form:

```
ObjectID@MethodName(Argument_1,...,Argument_n)
```

the Snart method despatcher first finds the object's class (by looking up sn_object for ObjectID), and then looks up a property of the form:

```
<ClassName,MethodName,Predicate>
```

If found, Snart then invokes the method predicate using:

```
Predicate(ObjectID,Argument_1,...,Argument_n)
```

If the predicate look-up failed, Snart tries to find a default method predicate common to all objects, and executes this if found. Otherwise Snart raises an exception saying the method is invalid for the object, and aborts the current Prolog process

The called method predicate can fail, in which case Snart just fails the method call and allows the calling predicate to take what ever action it desires (i.e. the method call fails in the same manner a Prolog predicate fails).

When the `create` method is called for an object, the object ID given is expected to be a variable. Snart firstly creates an object using `new_object` (see below). If a method predicate `ClassName::create` is defined for the new object's class, this is then called with the remaining arguments of create.

B.5.3. Other Object Predicates and Methods

`new_object(-ClassName,-ObjectID)`

> Creates a new object of type ClassName and returns the new object's ID as ObjectID. Aborts if ClassName is not a valid Snart class name.

`object_class(-ObjectID,?ClassName) and ObjectID@class(?ClassName)`

> Returns `<ObjectID,sn_object,ClassName>` property for the given object. Aborts if the object does not exist.

`copy_object(-ObjectID,+NewObjectID) and ObjectID@copy(+NewObjectID)`

> Calls `new_object` to create `NewObjectID` of same class as `ObjectID`. Copies all property values of `ObjectID` to `NewObjectID`. Aborts if ObjectID does not exist.

`member_class(-ObjectID,-ClassName) and ObjectID@member(-ClassName)`

> Gets object's class and does check of class membership using the parents the object's class. This check is recursive and fails if ClassName is not an ancestor of ObjectID. `member_class` is optimised by keeping a list of all visited classes during the search. This is because multiple, repeated inheritance can mean the object's class inherits from a class

more than once, and the search process need not re-visit a class and check its ancestors for ClassName if already checked once.

```
object_attribute(-ObjectID,-AttributeName) and
```
```
ObjectID@attribute(-AttributeName)
```

Gets a list of attributes for ObjectID's class using the attribute look-up table and succeeds if AttributeName is on this list. Three argument form returns the type of the attribute requested.

```
classify_object(-ObjectID,-Classifier,-NewClass) and
```
```
ObjectID@classify(-Classifier,-NewClass)
```

Looks up classifier attribute and changes object's class by changing `sn_object`. Also records `NewClass` as value of `Classifier` property for `ObjectID`. On re-classifying this object, attributes not compatible with old class are removed, then object re-classified. If a class has more than one classifier and this object has been classified previously, `classify_object` makes the object a member of both classes (if `NewClass` is not a descendant of the object's previous classification classes). Snart generates a new class of the form '`sn_merged([ClassName1,ClassName2,...])`'. The method and attribute despatch tables for this class are a union of the `ClassName1` and `ClassName2` tables. Aborts if the classification is invalid (e.g. `NewClass` not on the list of classes for Classifier).

```
is_class(-ClassName)
```

Checks if ClassName has a predicate of the form snc_ClassName (i.e. is a valid, compiled Snart class).

```
is_object(-ObjectID) and ObjectID@is_object
```

Looks up the `<ObjectID,sn_object,ClassName>` property for `ObjectID`. Succeeds if this property is found, fails otherwise.

B.5.4. Object Spying

Snart objects can be spied by `sn_trace_object(ObjectID[,FeatureNames])`. This generates events that equate to method entry/exit and attribute update by calling `sn_entry(ObjectID, Method(Arguments))`, `sn_exit(ObjectID, Method(Arguments), Success?)` and `sn_set_value(ObjectID, Attribute, OldValue, NewValue)`. Calling `sn_untrace_object(ObjectID[,FeatureNames])` will remove the event generation for the object or given features.

Object spying is implemented by classifying an object to a merged class made up of the object's orginal class and a special `spy` class. This `spy` class provides additional methods `sn_set_attribute` and `sn_despatch_method`. These are used in preference to the default method despatching and attribute updating by the Snart method despatcher if no despatch table entries for a method or attribute are found. Classification deletes the despatch table entries for classified classes and looks up values from the classes that form

the merged (union) class dynamically. Any `sn_set_attribute` and `sn_despatch_method` methods defined are used in preference to this default look-up. Spied features thus generate tracing events by having `sn_set_attribute` and `sn_despatch_method` generate the events and then call the method defined by a classifier class to maintain an object's orginal behaviour.

B.5.5. Object Persistency

Snart objects can be stored in a persistent object store and reloaded when accessed. Updated objects are rewritten to the store when it is closed or on demand. `sn_create_object_store(File,Path)` and `sn_open_object_store(File,Path)` create and open an object store respectively. Objects which inherit from a class `sn_persistent` are assumed to be persistent and behave as such. `sn_close_object_store(File)` writes any updated objects to the store and closes it. `sn_write_objects` updates the store without closing it. Currently only one object store is allowed to be open at one time and object stores can not be merged.

Persistent objects are implemented in a similar way to spyed objects with `persistent` providing additional meta-level methods `sn_alloc_id` and `sn_delete` for allocating and deleting persistent storage for a new object. `sn_set_attribute` marks an object as updated so `sn_write_objects` will save its state. An additional predicate `sn_find_object` is defined to reload an object from persistent storage when it is accessed for the first time (as its object data won't be in-core). Persistency and object spying are consistent as a spied, persistent object will first generate events and then perform persistency-management operations.

B.6. Performance

The object-oriented structure of Snart introduces an extra level of abstraction over conventional Prolog. Snart programs are structured around classes and data is stored as objects verses the conventional Prolog structure supported by LPA. We compare the performance of Snart to LPA Prolog in three ways:

- speed
- memory requirements
- program structuring, reusability, debugging and maintenance

B.6.1. Speed

Fig. B.9. illustrates the performance of Snart feature calls compared with Prolog predicate calls. This test used 100 calls to predicates that performed exactly the same processing. The Snart method despatcher must look-up the correct method predicate to call for an object (which involves getting the object's class and finding the correct method predicate

the the method call selector). One argument feature calls may be either an attribute fetch or a method call and Snart must determine whether to fetch an attribute value or call a method at run-time. Hence the extra time needed for one argument calls verses other kinds. Typically Snart feature calls incur a 40% execution time overhead compared to Prolog predicate calls. As Snart methods may call conventional Prolog predicates to do part of their processing, this overhead is often much less in Snart application programs.
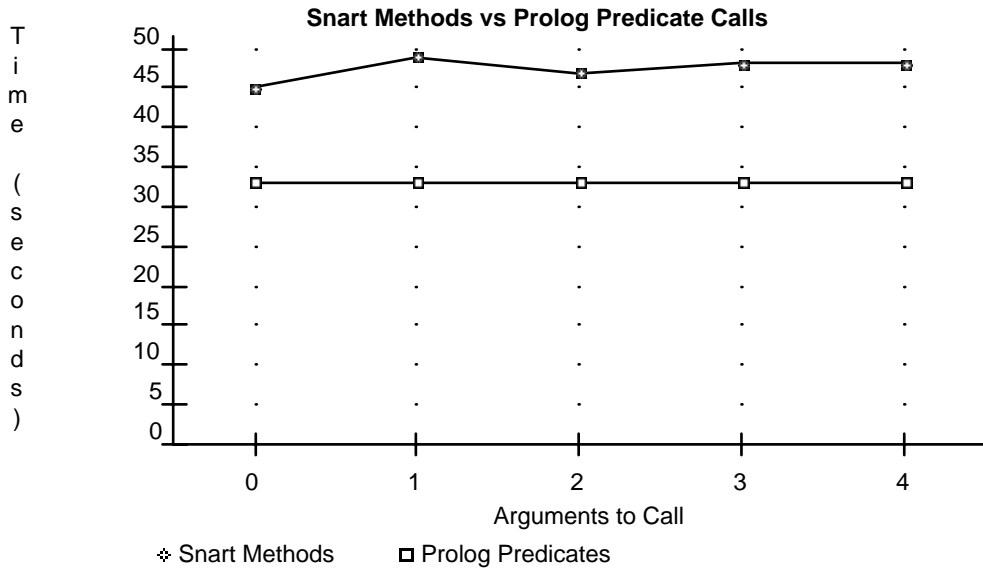
fig. B.9.   Snart method predicate calls vs. Prolog predicate calls.

Fig. B.10. compares attribute fetching and assignment in Snart to prolog programs using the Prolog database and LPA's property management predicates. This test used 200 accesses or assignments of the same data for each technique. Snart attribute fetches and assignment compare well to conventional LPA properties for getting and setting values. However, Snart is significantly slower if the attribute has not been assigned a value (Snart will try to find a method for the object if an attribute fetch on a one-argument feature call fails, hence the extra over-head). The Prolog database is most efficient for looking up attribute values (which have either been assigned or not). However, it is extremely slow for modifying an attribute value (as the old clause must be retracted and a new clause asserted).

Time for attribute access/setting
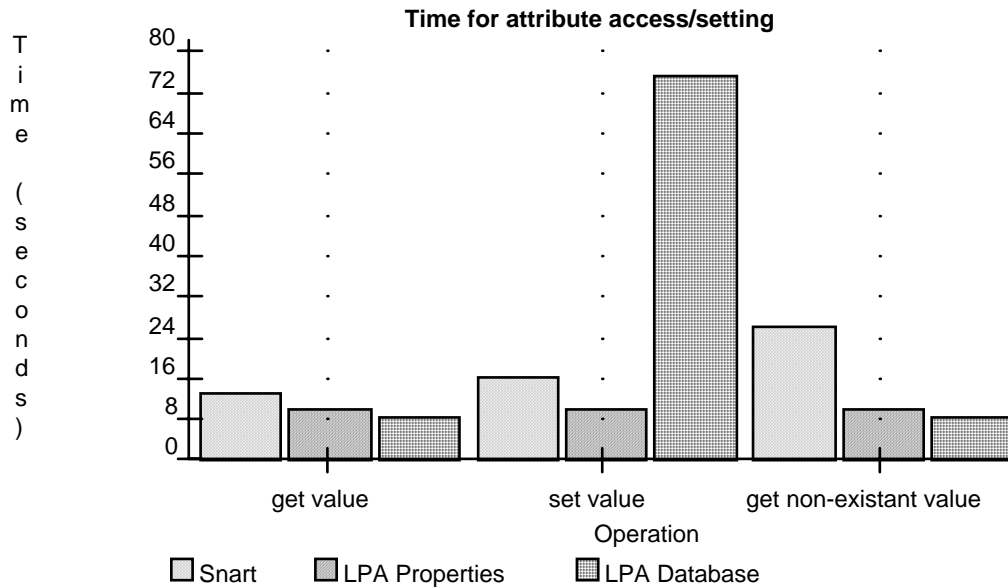
fig. B.10. Snart attribute access/assignment vs. Prolog database and LPA properties.

B.6.2. Memory Requirements

The memory requirements for programs using Snart objects and LPA properties are almost identical. When creating many small objects, a small over-head is incurred as Snart must store a class name against each object created. As property management in LPA is more efficient in both time and space than using the Prolog database to store values, Snart objects take up less space than similar programs using the database.

B.6.3. Programming in Snart vs. conventional LPA Prolog

Snart provides several improvements on raw LPA Prolog programming:
- program structuring around classes
- uniform treatment of data as objects (instances of classes)
- modification of classes does not necessarily invalidate object data
- reuse of programs via inheritance, type aggregation and client-server
- compile-time and run-time checking of classes for program and data consistency

As Snart is a hybrid language, Snart programs can make use of any conventional LPA Prolog program while using an object-oriented structure. This support for high-level structuring enhances Snart programs compared with equivalent Prolog programs in several ways:
- Snart programs are more easily modified, as changes to the class hierarchies are easier to identify than changes to Prolog predicates (especially when changing the way in which data is accessed and stored).

- Reuse of Snart programs is easier using inheritance and composition from existing classes.
- Snart classes can be modified while preserving their instances (which is very useful for debugging). Prolog programs using either the database or properties are seldom as easy to change (Grundy 91).
- Snart programs have a more well-defined structure and can be organised into frameworks and other building-blocks more naturally than conventional Prolog code.

B.7. Comparison to Other Object-Oriented Prologs

Most object-oriented extensions to Prolog, including Protalk (Quintus 91), Prolog++ (Pountain 90) and ObjVProlog (Malenfant et al 89) treat classes as objects. Classes are defined by creating instances of a "class" object, and objects by duplicating a class object or creating instances of it. Snart treats classes like Eiffel and C++: as implementations of abstract data types which are defined at compile-time. Thus Snart programs have a similar design philosophy to strongly-typed object-oriented languages and do not use concepts not readily portable to these languages.

Snart is a simple object-oriented language with classes being composed of parent and feature definitions. ObjVProlog and other meta-class based languages define meta-level classes which can in turn be specialised to provide new types of objects behaviour. Examples include persistent objects and parallel objects (Malenfant et al 89). Snart treats all feature calls the same with an identical syntax. Most other Prologs have different forms of attribute access and method calling.

Snart class hierarchies are fixed at run-time, whereas other object-oriented Prologs adopting the class-as-object approach can modify their set of classes as desired at run-time. The Snart approach follows the same approach as class-based languages such as C++ and Eiffel and is more appropriate for software engineering support for programs. Lack of run-time class creation does not appear to make Snart any less useful as a prototyping language in our experience.

Snart programs tend to be more easily maintained than Protalk programs which must either define class creation predicates or store class definitions separately to Prolog programs. Prolog++ programs define their class objects along with LPA Prolog code in program windows in the same manner as Snart.

The implementation of Snart compiles classes into a very compact form with method despatch tables being generated for each concrete class. Prolog++, Protalk and ObjVProlog have similar yet different methods for handling method despatch. Prolog++ keeps a list of super-classes which are searched on method look-up and the appropriate predicate called.

Protalk holds a predicate name which is called with an object ID, method selector and method call arguments. This predicate then determines the action to take on the method call. Attributes are accessed or set by calling the same predicate with an attribute name and value (a variable for attribute look-up). The Protalk approach is very slow and adds large over-heads for method calling over conventional Prolog predicate calling.

Prolog++ provides a number of compilation optimisations including first-term indexing and direct method calling of inherited features. It also provides daemon support for data and event-driven programming and information hiding. Snart could be extended to support all these facilities and optimisations, but some are not in the style of languages we may wish to implement Snart programs in.

Snart shares common object-oriented facilities with most other object-oriented Prologs. Snart is based on the C++ notion of a distinction between classes and objects, however, and hence is more suitable for implementing software and using as a representative object-oriented language than other Prologs we have seen.

B.8. Quintus Snart

We have ported Snart to Quintus Prolog (Quintus 91a) which runs on Unix systems. This port was done to illustrate Snart can be transferred to Prolog systems other than LPA MacProlog and can run on machines other than the Macintosh. We briefly describe the Quintus Prolog version of Snart and the differences between this version and the original Macintosh Snart.

B.8.1. Compiler

The Snart compiler is basically the same for both the LPA and Quintus versions. Quintus does not allow variable functor names which were quite extensively used in the Snart compiler. As Quintus provides better module facilities and other program structuring support, the Quintus version of the compiler is somewhat cleaner than the LPA version. Both compilers generate exactly the same compiled class definitions and method and attribute look-up tables.

B.8.2. Run-time System

The run-time systems for the two versions of Snart are significantly different. The Quintus version currently uses the Prolog database to store object information, as Quintus does not provide the property management facilities of LPA. Quintus also does not provide as flexible predicate calling functions as LPA so the method despatcher is somewhat more complicated. Both versions support the same object manipulation method and predicate calls. To make the Quintus version perform better in terms of run-time speed (particularly

for attribute updating) we could implement a property management system similar to that of LPA Prolog.

B.8.3. Environment

As Quintus Prolog uses a command-line environment, Snart provides access to its internal predicates for compiler invocation and object manipulation. This allows programmers to compile classes by invoking the predicate `sn_compile_classes` and to display objects by calling `ObjectID@print`. The Quintus environment for Snart is not nearly as natural to use as the LPA Prolog version. We could extend this to provide similar window-based facilities by extending Quintus's X-windows based environment to provide pull-down menus for Snart similar to those provided by LPA.

B.9. Future Extensions

B.9.1. Explicit Redefinition and Information Hiding

Snart allows any feature inherited from a parent to be redefined in a class by simply defining a new feature of the same name. Snart currently supports no information hiding with all features being visible and accessible outside an object.

To provide explicit redefining is a very simple compilation check to ensure features redefined have a `redefine(FeatureName)` entry in the rename list associated with a parent class. To support information hiding run-time checks must be included to ensure access to a private or protected feature is valid. To do this, the Snart compiler must inspect all predicates (including method predicates) and associate a "called by" class with every feature call. The method despatcher can then check whether methods and attributes are being accessed correctly for the object being sent the message (as part of the feature look-up process). Features accessed incorrectly can then cause the current Prolog process to abort and report the error.

B.9.2. Data-driven Support

Data and event-driven programs are difficult to write in Snart without providing explicit methods that set/get object attribute values and record updates to objects (as done for the MViews framework described in Chapter 7). The disadvantage of this approach is that attribute updates and other updates to an object must be explicitly catered for when designing and implementing a program. Subsequent modification of the program may require additional updates to be reported and hence modification of existing classes to provide for the needs of these changes. Chapter 7 compares language-based and framework-based support for data-driven programs in further detail.

To extend Snart to allow data-driven programs based on attribute changes is relatively simple. A predicate can be associated with each class (or even each attribute of a class) which is called when the attribute is set via the `:=` operator. This can be built into Snart with no loss of run-time efficiency, even for predicates with no need for an "update" predicate by changing the attribute parameter of the `ClassName(AttributeName,attribute,Type)` predicates in the look-up table for attributes. The new parameter is the predicate to call to perform for attribute of the object's attribute (set_value at present). This predicate and any dependent objects that are sent messages on the attribute update could be implemented to provide a similar facility to the Smalltalk model-view relationship (Goldberg and Robson 84).

To extend Snart to notify other objects when an object's methods are invoked or exit is somewhat more difficult. This type of "update" is useful when complex changes are made (e.g. an element is added to, removed from or simply moved within a list attribute) and the kind of change is important, not just the fact that an update has occurred. MViews requires these updates to be explicitly determined to record the kind of change (for storing against base elements, updating subset view elements, redrawing display elements and providing a generic undo/redo facility). Trapping every object method call, as done in the Cerno debugger for tracing Snart programs (Fenwick and Hosking 93), is prohibitively expensive. We could provide a facility similar to that for the Snart debugger (see Chapter 9) which causes "features of interest" to call predicates with their arguments before and after execution.

B.9.3. Optimisations for Performance Enhancement

The run-time performance of Snart could be enhanced by performing several optimizations when compiling Snart programs. Calls to renamed, inherited features could be converted into direct predicate calls. For example, in the example in fig. B.11., we can change the compiled code to that on the bottom right.

```
abstract_class(figure,              % Unoptimized code:
  parents([]),
  features([create,...])).          'rectangle::create'(Rectangle,Location,
                                        Width,Height) :-
figure::create(Figure,Location) :-   Rectangle@fig_create(Location),
  Figure@location:=Location,          Rectangle@width:=Width,
  Figure@visible:=false.              Rectangle@height:=Height.

class(rectangle,                    % Optimized code:
 parents([figure(
   [rename(create,fig_create)])]),  'rectangle::create'(Rectangle,Location,
 features([create,...])).              Width,Height) :-
                                       'figure::create'(Ractangle,Location),
rectangle::create(Rectangle,Location, Rectangle@width:=Width,
   Width,Height) :-                    Rectangle@height:=Height.
  Rectangle@fig_create(Location),
  Rectangle@width:=Width,
  Rectangle@height:=Height.
```

fig. B.11. Optimizing method despatch for Snart.

Snart can perform a simple optimization of its method and attribute look-up tables by compiling them using an optimized LPA program window. Method predicates can also be compiled using an optimized program window to give first-term indexing and other optimizations. To allow for first-term indexing, however, Snart must not have the first argument of method predicates being a variable for the ID of an object. We should thus move this default variable to being, for example, the last argument of a method predicate.

If we extend Snart to allow for typed variables for object ID's (see below), we can also optimize the method despatcher at compile-time and perform various type checks (e.g. to support information hiding and check for existing features at compile-time). Note that all these optimizations require the Snart compiler to look at every predicate call in a term to ensure that the appropriate checks and optimizations are performed. As this may be quite a large overhead, optimizations should only be performed on user request.

B.9.4. Typed Variables

Adding typed variables to Snart also requires compile-time checking of all terms in a Snart program to remove the typing information for asserted predicates. Fig. B.12. shows a 'typed' version of the predicates in fig. B.11.

```
abstract_class(figure,                class(rectangle,
  parents([]),                         parents([figure(
  features([create,...])).               [rename(create,fig_create)])]),
                                        features([create,...])).
figure::create(Figure:figure,Location) :-
  Figure@location:=Location,          rectangle::create(Rectangle:rectangle,
  Figure@visible:=false.                  Location,Width,Height) :-
                                        Rectangle@fig_create(Location),
                                        Rectangle@width:=Width,
                                        Rectangle@height:=Height.
```

fig. B.12. Typed Snart variables.

As the general class of an object can now be determined at compile-time, the Snart compiler could make any checks for incorrectly accessing private features outside a class, accessing nonexistent features for a class and type mis-matches (e.g. adding figure objects to a list of rectangle objects). Note that we must strip all this extra information about types from terms before asserting them as it will confuse Prolog's unification algorithm (an integer object ID will not match an argument of the form Figure:figure).

B.9.5. Lazy, Functional Feature Evaluation

Snart currently provides support for object-oriented, logic programming (with an imperative flavour being added by assignment to object attributes). It would also be useful to provide lazy, functional evaluation of object features (functions) in a similar manner to Kea (Hosking et al 90).

Snart could be extended to declare "functional" methods and attributes, as shown in fig. B.13. `area` and `volume` are only re-evaluated when one of their dependent values is changed (by attribute assignment or re-evaluation).

```
class(rectangle,                          class(box,
 parents([figure(                           parents([]),
   [rename(create,fig_create)])]),          features([base:rectangle,depth:integer,
 features([width:integer,height:integer       functional volume,...])).
   functional area,...])).

rectangle::area(Rectangle,Area) :-       box::volume(Box,Volume) :-
  Area     is    Rectangle@width    *     Volume is Box@base@area * Box@depth.
Rectangle@height.
```

fig. B.13. Lazy, functional features in Snart.

These features are evaluated in a lazy fashion, so they are not executed until their value is actually required. We could then, for example, declare structures that are recursive and hence (theoretically) infinite, but since their values are only evaluated when needed, only take up as much space as required. In addition, we gain language-based support for such concepts as attribute grammars (data-driven evaluation of language semantics) (Reps and Teitelbaum 87) and tool-based abstraction using functional dependencies (Kaiser et al 92).

To implement functional features in Snart would require a similar mechanism to Kea, where various dependencies are maintained between functional and possibly non-functional attributes. A change propagation algorithm would re-evaluate any attributes whose value depended on one or more changed values.

Appendix C

A Gofer Implementation of MVSL

--
-- MVSL Abstract Syntax Definition
--

```
infix   2 :=
infixl  1 :&

type Ide = String

data Program = Pro [Decl] Command

-- Declarations
--
data Decl = BaseView Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            BaseElement Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            BaseRelationship Ide ParentDecl ChildDecl [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            SubsetView Ide ComponentDecl [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            SubsetElement Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            SubsetRelationship Ide ParentDecl ChildDecl [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            Component Ide [AttributeDecl] [RelationshipDecl] [OperationDecl] [UpdateDecl] |
            Global Ide Type

data ComponentDecl = Components [Ide]
data ParentDecl = Parent Type
data ChildDecl = Child Type
data AttributeDecl = Attribute Ide Type
data RelationshipDecl = Relationship Ide Type
data OperationDecl = Operation Ide [OpArgumentDecl] [LocalDecl] Command |
          Function Ide [OpArgumentDecl] Type [LocalDecl] Command
data UpdateDecl = Update Ide [LocalDecl] Exp [LocalDecl] Command

data OpArgumentDecl = InArg Ide Type | OutArg Ide Type
data LocalDecl = Arg Ide Type

-- Commands & Operations
--
data Command = Exp := Exp |
          Eskip |
          Eifthen Exp Command Command |
          Ewhile Exp Command |
          Eforall Exp Exp Command |
          EWrite Exp |
          Command :& Command |
          AddElement Ide Exp |
          DeleteComponent Exp |
          Establish Type Exp Exp Exp |
          EstablishLink Type Exp Exp |
          Reestablish Exp Exp Exp |
          Disolve Type Exp Exp |
          Record Exp Ide [Exp] |
          Store Exp Ide [Exp] |
          CreateView Ide Exp |
          AddViewComponent Exp Exp |
          RemoveViewComponent Exp Exp |
          ApplyOp Exp [Exp]
```

```
-- Expressions
--
data Exp = IntLit Int |
          StringLit String |
          True_ | False_ |
          Ident String |
          Op Opr Exp Exp |
          CompVal Exp Ide |
          FuncOp Exp [Exp]

data Opr = Plus | Minus | Times | Divide | Gt | Lt | Eq | Neq | And | Or | Append | Remove


-- Types
--
data Type = BoolType |
           StringType |
           IntType |
           ListType Type |
           OneToOne Ide |
           OneToMany Ide |
           ComponentType Ide |
           CompAttrType Ide Ide
--
-- MVSL Declarations
--
-- Given a list of program declarations, produce a type-map
--

-- DeclValue maps type identifier to type value
--
type DeclValue = Ide -> TypeValue

emptyDeclValue :: DeclValue
emptyDeclValue _ = TNotDefined

updateDeclValue :: DeclValue -> Ide -> TypeValue -> DeclValue
updateDeclValue dv n tv i = if i==n then tv else dv i

-- BasicKind gives the basic kinds of MViews components
--
data BasicKind = KBaseView | KBaseEl | KBaseRel |
                KSubsetView | KSubsetEl | KSubsetRel |
                KComp | KLinkRel

-- TypeValue stores type for an identifier
--
-- Type values may be integers, lists, components, etc. or the whole
-- value for a component declaration.
--
-- A component's basic kind,attribute values (which must be allocated
-- when the component is created) and the type of each component value
-- (attributes, relationships, operations, etc.)
--
data TypeValue = TCompData BasicKind [Ide] CompTypes |
                TString |
                TInteger |
                TBool |
                TList TypeValue |
                TOneToOne Ide |
                TOneToMany Ide |
                TComp Ide |
                TCompAttr Ide Ide |
                TVoid |
                TAny |
                TNotDefined

instance Eq TypeValue where
        TString == TString = True
        TInteger == TInteger = True
        TBool == TBool = True
        TList t1 == TList t2 = t1 == t2
        TComp c1 == TComp c2 = c1 == c2
```

```
              TAny == _ = True
              _ == TAny = True
              _ == _ = False

-- CompTypes maps identifiers to their component type
--
type CompTypes = Ide -> CompType

emptyCompTypes :: CompTypes
emptyCompTypes _ = CNotDefined

updateCompTypes :: CompTypes -> Ide -> CompType -> CompTypes
updateCompTypes ct n v i = if i==n then v else ct i

-- A Component type is one of attribute, relationship, etc.
--
-- parent, child and components for relationships and views are stored
-- as CAttribute
--
data CompType = CAttribute TypeValue |
        CRelationship TypeValue |
        COperation OpArgs TypeValue OpLocals CommandMeaning |
        CUpdates [CUpdate] |
        CNotDefined

-- Updates may have >1 value (e.g. several update_attributes for
-- different types and guards).
--
data CUpdate = UpdateOp OpArgs OpLocals ExpMeaning CommandMeaning

-- Arguments are in or out
--
type OpArgs = [(Ide,InOrOut,TypeValue)]
data InOrOut = In | Out

updateOpArgs :: OpArgs -> Ide -> InOrOut -> TypeValue -> OpArgs
updateOpArgs oa n io t = oa++[(n,io,t)]

-- Local variables have identifier and type
--
type OpLocals = [(Ide,TypeValue)]

updateOpLocals :: OpLocals -> Ide -> TypeValue -> OpLocals
updateOpLocals ol n t = ol++[(n,t)]

-- Operations and update operations have a meaning given a State
-- (i.e. given one state produce another - see below for State etc.)
--
-- Update operation guards have a Value given a State (see below)
--
type CommandMeaning = (State -> Command -> State) -> State -> State
type ExpMeaning = (State -> Exp -> Value) -> State -> Value

-- Compute the declarations value for a list of program declarations
--
-- Returns DeclValue for program and a list of identifiers
-- to create locations for (i.e. globals)
--
-- rel_comps computes any link relationships defined by the component
-- and adds a DeclValue for their name (name = CompKind.RelName)
--
program_decls :: [Decl] -> DeclValue -> [Ide] -> (DeclValue,[Ide])
program_decls [] dv gs = (dv,gs)
program_decls (d:ds) dv gs = (new_dv,new_gs) where
        (n,tv,rs,globals) = decl_value d
        (new_dv,new_gs) = program_decls ds
                (updateDeclValue (rel_comps dv rs n)  n tv) (gs++globals)

-- Compute the Ide/TypeValue/Link Relationships/Globals for a declaration
--
-- For each basic component kind the CompTypes value is computed, including the
-- attributes it defines (so space for these can be allocated when the
```

-- component is created), any default values for the component (see below),
-- and its declaration values are processed to produce a CompTypes value for each
-- identifier.
--
decl_value :: Decl -> (Ide,TypeValue,[RelationshipDecl],[Ide])
decl_value (BaseView name as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KBaseView (update_types (op_types (rel_types (attribute_types
                ([],emptyCompTypes) as) rs) os) us)
        tv = (TCompData KBaseView names comp_ts)
decl_value (BaseElement name as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KBaseEl (update_types (op_types (rel_types (attribute_types
                ([],emptyCompTypes) as) rs) os) us)
        tv = (TCompData KBaseEl names comp_ts)
decl_value (BaseRelationship name pd cd as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KBaseRel (update_types (op_types (rel_types (attribute_types
                (parent_types (child_types ([],emptyCompTypes) cd) pd) as) rs) os) us)
        tv = (TCompData KBaseRel names comp_ts)
decl_value (SubsetView name els as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KSubsetView (update_types (op_types (rel_types (attribute_types
                (comp_types ([],emptyCompTypes) els) as) rs) os) us)
        tv = (TCompData KSubsetView names comp_ts)
decl_value (SubsetElement name as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KSubsetEl (update_types (op_types (rel_types (attribute_types
                ([],emptyCompTypes) as) rs) os) us)
        tv = (TCompData KSubsetEl names comp_ts)
decl_value (SubsetRelationship name pd cd as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KSubsetRel (update_types (op_types (rel_types (attribute_types
                (parent_types (child_types ([],emptyCompTypes) cd) pd) as) rs) os) us)
        tv = (TCompData KSubsetRel names comp_ts)
decl_value (Component name as rs os us) = (name,tv,rs,[]) where
        (names,comp_ts) = default_types KComp (update_types (op_types (rel_types (attribute_types
                ([],emptyCompTypes) as) rs) os) us)
        tv = (TCompData KComp names comp_ts)
decl_value (Global name t) = (name,type_value t,[],[name])

-- Compute link relationships for a component
--
rel_comps :: DeclValue -> [RelationshipDecl] -> Ide -> DeclValue
rel_comps dv [] comp_name = dv
rel_comps dv ((Relationship name (OneToOne _)):rs) comp_name = new_dv where
        new_dv = rel_comps (updateDeclValue dv (comp_name++"."++name) link_rel) rs comp_name
rel_comps dv ((Relationship name (OneToMany _)):rs) comp_name = new_dv where
        new_dv = rel_comps (updateDeclValue dv (comp_name++"."++name) link_rel) rs comp_name

link_rel :: TypeValue
link_rel = TCompData KLinkRel [] emptyCompTypes

-- Compute parent declaration for component
--
parent_types :: ([Ide],CompTypes) -> ParentDecl -> ([Ide],CompTypes)
parent_types (names,ct) (Parent t) =
        (["parent"]++names,updateCompTypes ct "parent" (CAttribute (type_value t)))

-- Compute child declaration for component
--
child_types :: ([Ide],CompTypes) -> ChildDecl -> ([Ide],CompTypes)
child_types (names,ct) (Child t) =
        (["child"]++names,updateCompTypes ct "child" (CAttribute (type_value t)))

-- Compute component types for subset view
--
comp_types :: ([Ide],CompTypes) -> ComponentDecl -> ([Ide],CompTypes)
comp_types (names,ct) (Components comps) =
        (["components"]++names,updateCompTypes ct "components" (CAttribute (TList TAny)))

-- Compute attribute types for list of attribute declarations
--
attribute_types :: ([Ide],CompTypes) -> [AttributeDecl] -> ([Ide],CompTypes)
attribute_types (names,ct) [] = (names,ct)
attribute_types (names,ct) ((Attribute n t):as) =
        attribute_types ([n]++names,(updateCompTypes ct n (CAttribute (type_value t)))) as

---

```
-- Compute relationship types for list of relationship declarations
--
rel_types :: ([Ide],CompTypes) -> [RelationshipDecl] -> ([Ide],CompTypes)
rel_types (names,ct) [] = (names,ct)
rel_types (names,ct) ((Relationship n t):rs) =
        rel_types ([n]++names,(updateCompTypes ct n (CRelationship (type_value t)))) rs

-- Compute operation types for list of operation declarations
--
op_types :: ([Ide],CompTypes) -> [OperationDecl] -> ([Ide],CompTypes)
op_types (names,ct) [] = (names,ct)
op_types (names,ct) ((Operation n arg_decls loc_decls command):os) =
        op_types ([n]++names,(updateCompTypes ct n (op_value arg_decls loc_decls (op_meaning command)))) os
op_types (names,ct) ((Function n arg_decls t loc_decls command):os) =
        op_types ([n]++names,(updateCompTypes ct n (fn_value arg_decls t loc_decls (op_meaning command)))) os

-- Compute update types for list of update declarations
--
-- This produces a list of guarded, input-only operations which are
-- event-driven by updates on a component.
--
update_types :: ([Ide],CompTypes) -> [UpdateDecl] -> ([Ide],CompTypes)
update_types (names,ct) [] = (names,ct)
update_types (names,ct) ((Update n arg_decls guard loc_decls command):us) = nt where
        upd_op = update_value arg_decls loc_decls (exp_meaning guard) (op_meaning command)
        nt = case (ct n) of
                (CUpdates updates) ->
                  update_types (names,(updateCompTypes ct n (CUpdates (updates++[upd_op])))) us
                CNotDefined ->
                  update_types ([n]++names,(updateCompTypes ct n (CUpdates [upd_op]))) us

-- "Meanings" for operations and expressions
--
op_meaning :: Command -> (State -> Command -> State) -> State -> State
op_meaning c fn s = fn s c

exp_meaning :: Exp -> (State -> Exp -> Value) -> State -> Value
exp_meaning e fn s = fn s e


-- Value of an operation declaration
--
-- Defined as its argument's types and in/out status, its local's types
-- and the meaning of its associated command
--
op_value :: [OpArgumentDecl] -> [LocalDecl] -> CommandMeaning -> CompType
op_value as ls command = (COperation (op_arg_types as) TVoid (local_types ls) command)

-- Value of a "functional operation" is same as for operation but with a type
--
fn_value :: [OpArgumentDecl] -> Type -> [LocalDecl] -> CommandMeaning -> CompType
fn_value as t ls command = (COperation
        (op_arg_types as) (type_value t) (updateOpLocals (local_types ls) "result" (type_value t)) command)

-- Value of an "update operation" is same for operation but arguments are input-only
--
update_value :: [LocalDecl] -> [LocalDecl] -> ExpMeaning -> CommandMeaning -> CUpdate
update_value as ls guard command = (UpdateOp (update_arg_types as) (local_types ls) guard command)

-- Bind operation arguments to in/out status and type
--
op_arg_types :: [OpArgumentDecl] -> OpArgs
op_arg_types [] =[]
op_arg_types ((InArg n t):as) =
        updateOpArgs (op_arg_types as) n In (type_value t)
op_arg_types ((OutArg n t):as) =
        updateOpArgs (op_arg_types as) n Out (type_value t)

-- Bind local variables to type
--
local_types :: [LocalDecl] -> OpLocals
local_types [] = []
local_types ((Arg n t):as) =
```

```
                updateOpLocals (local_types as) n (type_value t)

-- Bind update arguments to type
--
update_arg_types :: [LocalDecl] -> OpArgs
update_arg_types [] = []
update_arg_types ((Arg n t):as) = updateOpArgs (update_arg_types as) n In (type_value t)


-- Value of a type
--
type_value :: Type -> TypeValue
type_value (BoolType) = (TBool)
type_value (StringType) = (TString)
type_value (IntType) = (TInteger)
type_value (ListType t) = (TList (type_value t))
type_value (OneToOne c) =(TOneToOne c)
type_value (OneToMany c) = (TOneToMany c)
type_value (ComponentType n) = (TComp n)
type_value (CompAttrType c a) = (TCompAttr c a)


-- Default attributes and component types for a component given its "BasicKind"
--
data DefaultType = Default Ide CompType

default_types :: BasicKind -> ([Ide],CompTypes) -> ([Ide],CompTypes)
default_types _ cts =
        addCompTypes [
                (Default "class" (CAttribute TString)),
                (Default "relationships" (CAttribute (TList TAny))),
                (Default "updates" (CAttribute (TList TAny)))] cts

addCompTypes :: [DefaultType] -> ([Ide],CompTypes) -> ([Ide],CompTypes)
addCompTypes [] (names,ct) = (names,ct)
addCompTypes ((Default n t):ds) (names,ct) = addCompTypes ds ([n]++names,(updateCompTypes ct n t))


--
-- MViews state definitions
--


-- A CompStore is used to record the attribute values for a component.
-- A distinguished attribute "class" gives the component kind for
-- a component instance.
--
type CompID = Int
type CompStore = CompID -> Ide -> CompValue
data CompValue = NoCValue | CValue Dv

emptyCompStore :: CompStore
emptyCompStore _ _ = NoCValue

new_comp :: CompStore -> Ide -> (CompStore,CompID)
new_comp s k = new_comp' 1 where
        new_comp' i = case s i "class" of
                NoCValue -> (updateCompStore s i "class" (Rv (Vstring k)),i) ; _ -> new_comp' (i+1)

updateCompStore :: CompStore -> CompID -> Ide -> Dv -> CompStore
updateCompStore s c a v i j = if i==c && j==a then (CValue v) else s i j

remove_comp :: CompStore -> CompID -> CompStore
remove_comp s c i j = if i==c then NoCValue else s i j

-- Relationships have parent/child attribute values in CompStore
--
is_rel :: State -> CompID -> Bool
is_rel s r = rel_kind where
        (CValue (Rv (Vstring kind))) = comps s r "class"
        (TCompData bk names cts) = declarations s kind
        rel_kind = case bk of
                KBaseRel -> True
                KSubsetRel -> True
                KLinkRel -> True
                _ -> False
```

```
-- All relationships for a component are stored by "relationships"
--
comp_rels :: State -> CompID -> [CompID]
comp_rels s c = rels where
        rels = case comps s c "relationships" of
                (CValue (Rv (Vlist rel_values))) -> values_to_comps rel_values
                _ -> []

-- View components for a view are stored in "components"
--
view_comps :: State -> CompID -> [CompID]
view_comps s c = vcomps where
        (CValue (Rv (Vlist comp_values))) = comps s c "components"
        vcomps = values_to_comps comp_values

values_to_comps :: [Value] -> [CompID]
values_to_comps [] = []
values_to_comps ((Vcomp c):vs) = [c]++values_to_comps vs

-- Denotable values
--
data Dv = Loc Location | Rv Value | CompValue CompID Ide

-- Expressable values
--
data Value = Vnum Int | Vbool Bool | Vstring String | Vcomp CompID | Vlist [Value] | Nil

-- instance of == for Value
--

instance Eq Value where
        (Vnum a) == (Vnum b) = a == b
        (Vbool a) == (Vbool b) = a == b
        (Vstring a) == (Vstring b) = a == b
        (Vlist a) == (Vlist b) = same_list a b
        (Vcomp a) == (Vcomp b) = a == b
        _ == _ = False

same_list [] [] = True
same_list (x:xs) (y:ys) = x == y && same_list xs ys


--
-- Store/Location for state variables
--

type Location = Int
type Store = Location->ValueOrUnused
data ValueOrUnused = Used Value | Unused

-- allocate new location in Store
--
new :: Store -> Location
new s = new' 0 where
        new' i = case s i of Unused -> i ; _ -> new' (i+1)

updateStore :: Store -> Location -> Value -> Store
updateStore s l v i = if i == l then Used v else s i

deallocStore :: Store -> Location -> Store
deallocStore s l i = if i==l then Unused else s i

emptyStore :: Store
emptyStore _ = Unused

-- return a list of n free locations from store
--
news :: Int -> Store -> ([Location],Store)
news n s = news1 n [] s where
        news1 0 ls s = (ls,s)
        news1 (n+1) ls s = news1' where
                l = new s
```

```
                  news1' = news1 n (l:ls) (updateStore s l Nil)


--
-- Environment for state variables
--
type Env = Ide -> ValueOrUnbound
data ValueOrUnbound = Bound Dv | Unbound

updateEnv :: Env -> Ide -> Dv -> Env
updateEnv e "" v i = e i
updateEnv e id v i = if i == id then Bound v else e i

deallocEnv :: Env -> Ide -> Env
deallocEnv e id i = if i == id then Unbound else e i

emptyEnv :: Env
emptyEnv _ = Unbound


-- Given a list of names and locations, bind names to locations in environment
--
extendEnv :: [Ide] -> [Location] -> Env -> Env
extendEnv [] [] e = e
extendEnv (n:ns) (l:ls) e = updateEnv (extendEnv ns ls e) n (Loc l)

-- Update records are a "term" of form Kind(Value1,Value2,...).
-- Outputs are just update records on a component.
--
data UpdateRecord = UpdateRec Ide [Value]
data Output = OV CompID UpdateRecord


--
-- The MViews program state is a tuple with component and location stores, an environment
-- and output list.
-- State also stores the DeclValue for a program as operations and updates must be
-- despatched on a per-component basis (could pass this value to all functions using
-- State, but its easiest to put it here).
--

type State = (CompStore,Env,Store,[Output],DeclValue)

emptyState :: DeclValue -> State
emptyState dv = (emptyCompStore,emptyEnv,emptyStore,[],dv)

-- State update functions
--
update_comps :: State -> CompStore -> State
update_comps (_,e,s,o,dv) c = (c,e,s,o,dv)

update_env :: State -> Env -> State
update_env (c,_,s,o,dv) e = (c,e,s,o,dv)

update_store :: State -> Store -> State
update_store (c,e,_,o,dv) s = (c,e,s,o,dv)

update_output :: State -> [Output] -> State
update_output (c,e,s,_,dv) o = (c,e,s,o,dv)

-- State access functions
--
comps :: State -> CompStore
comps (c,_,_,_,_) = c

env :: State -> Env
env (_,e,_,_,_) = e

store :: State -> Store
store (_,_,s,_,_) = s

output :: State -> [Output]
output (_,_,_,o,_) = o

declarations :: State -> DeclValue
```

```
declarations (_,_,_,_,dv) = dv

--
-- MVSL Commands
--
-- Includes:
--   :=
--   if-then
--   while
--   forall
--   Command ; Command
--   Comp.Operation(...args...)
--

-- Meaning of all commands is the meaning of a basic command
-- or the meaning of a component-specific operation.
--
command_meaning :: State -> Command -> State
command_meaning s c@(l := r) = assign s c
command_meaning s c@(Eskip) = s
command_meaning s c@(Eifthen e c1 c2) = if_then s c
command_meaning s c@(Ewhile e com) = while s c
command_meaning s c@(Eforall v e com) = for_all s c
command_meaning s c@(EWrite e) = write s c
command_meaning s c@(c1 :& c2) = sequence s c
command_meaning s c = operation_command s c

-- Assignment of the form lvalue := rvalue
--
assign :: State -> Command -> State
assign s (lexp := rexp) = assign_result s lv rv where
        lv = exp_val s lexp
        rv = rval s (exp_val s rexp)

-- The effect of assign_result is to change the value of a variable
-- or change a component attribute value. For the second case, an
-- update record is generated.
--
assign_result :: State -> Dv -> Value -> State
assign_result s (Loc l) rvalue = update_store s (updateStore (store s) l rvalue)
assign_result s (CompValue c a) rvalue = new_s where
        (CValue (Rv old_v)) = (comps s) c a
        assigned_s = update_comps s (updateCompStore (comps s) c a (Rv rvalue))
        new_s = update_dependents assigned_s c (UpdateRec "update_attribute" [Vcomp c,Vstring a,old_v,rvalue])

-- Conditional execution of the form
-- if <expression> then <command-if-true> else <command-if-false> end if
--
if_then :: State -> Command -> State
if_then s (Eifthen expr if_command else_command) = new_s where
        (Vbool ev) = rval s (exp_val s expr)
        new_s = if ev then command_meaning s if_command
                    else command_meaning s else_command

-- Conditional looping of the form
-- while <expression-true> to <command> end while
--
while :: State -> Command -> State
while s c@(Ewhile expr command) = new_s where
        (Vbool ev) = rval s (exp_val s expr)
        new_s = if ev then while (command_meaning s command) c
                    else s

-- List iteration of the form
-- forall <variable> on <list> do <command> end forall
--
for_all :: State -> Command -> State
for_all s (Eforall var expr command) = do_forall e s v command where
        v = exp_val s var
        (Vlist e) = rval s (exp_val s expr)
        do_forall [] s v command = s
        do_forall (x:xs) s v command = new_s where
```

```
                    v_s = assign_result s v x
                    c_s = command_meaning s command
                    new_s = do_forall xs c_s v command

-- Write command (for debugging)
--
-- Generates an "update record" which outputs an expression value
--
write :: State -> Command -> State
write s (EWrite e) = new_s where
            v = rval s (exp_val s e)
            new_s = update_output s ((output s)++[OV 0 (UpdateRec "write" [v])])

-- Sequentional commands of the form
-- <command1> <command2> ... <commandn>
--
sequence :: State -> Command -> State
sequence s (c1 :& c2) = command_meaning (command_meaning s c1) c2


--
-- The meaning of MViews Basic Operations
--
-- Defines:
-- update_attribute(in CompiD,in AttributeName,in NewValue)
-- add_element(in Kind,out CompID)
-- delete_element(in CompID)
-- establish(in Kind,in Parent,in Child,out NewRelID)
-- reestablish(in RelID,in NewParent,in NewChild)
-- disolve(in Kind,in Parent,in Child)
-- create_view(in Kind,out ViewID)
-- add_view_element(in ViewID,out CompID)
-- remove_view_element(in ViewID, in CompID)
-- record(in CompID,in Kind,in [Value])
-- store(in CompID,in Kind,in [Value])
-- Comp.Op([Value])
--

-- Meaning of basic operation "commands"
--
operation_command :: State -> Command -> State
operation_command s c@(Record e k u) = record_update s c
operation_command s c@(AddElement k e) = add_element s c
operation_command s c@(DeleteComponent e) = delete_component s c
operation_command s c@(Establish kind p ch v) = establish_rel s c
operation_command s c@(EstablishLink kind p ch) = establish_link_rel s c
operation_command s c@(Reestablish r p ch) = reestablish_rel s c
operation_command s c@(Disolve k p ch) = disolve_rel s c
operation_command s c@(CreateView k v) = create_view s c
operation_command s c@(AddViewComponent v e) = add_view_component s c
operation_command s c@(RemoveViewComponent v e) = remove_view_component s c
operation_command s c@(Store e k u) = store_update s c
operation_command s c@(ApplyOp e args) = apply_operation s c

-- add_element(in Kind,out CompID)
--
add_element :: State -> Command -> State
add_element s (AddElement kind new_var) = new_s where
            (comp_s,new_c) = add_component s kind
            (Loc new_loc) = exp_val comp_s new_var
            new_s = update_store comp_s (updateStore (store comp_s) new_loc (Vcomp new_c))

-- Add a new component and set given variable to the new component ID
--
add_component :: State -> Ide -> (State,CompID)
add_component s kind = (new_s,new_c) where
            (new_comps,new_c) = new_comp (comps s) kind
            (TCompData bk vs ct) = declarations s kind
            alloc_attributes [] cs c ct = cs
            alloc_attributes (n:ns) cs c ct =
                    case (ct n) of
                            (CAttribute t) ->
                                    if n == "class" then alloc_attributes ns cs c ct
```

```
                                        else updateCompStore (alloc_attributes ns cs c ct) c n (Rv Nil)
                        _ -> alloc_attributes ns cs c ct
        new_s = update_comps s (alloc_attributes vs new_comps new_c ct)

-- delete_element(in CompID)
--
delete_component :: State -> Command -> State
delete_component s (DeleteComponent exp) = new_s where
        (Vcomp c) = rval s (exp_val s exp)
        new_s = do_delete_component s c

do_delete_component :: State -> CompID -> State
do_delete_component s c = new_s where
        updated_s = update_dependents s c (UpdateRec "delete_element" [Vcomp c])
        dissolved_r = disolve_relationships (comp_rels s c) updated_s
        new_s = delete_comp dissolved_r c

-- Disolve all relationships to component
--
disolve_relationships [] s = s
disolve_relationships (r:rs) s =
        disolve_relationships rs (do_disolve_rel s r)

-- Remove all component data from state
--
delete_comp :: State -> CompID -> State
delete_comp s c = new_s where
        removed_view = remove_from_view s c
                -- remove from owning view (if any)
        new_s = update_comps removed_view (remove_comp (comps removed_view) c)

-- establish_rel(in Kind,in Parent,in Child,out NewRel)
--
establish_rel :: State -> Command -> State
establish_rel s (Establish kind parent child new_rel) = new_s where
        rk = rel_kind_type kind
        (Vcomp p) = rval s (exp_val s parent)
        (Vcomp c) = rval s (exp_val s child)
        (comp_s,new_r) = do_establish_rel s rk p c
        (Loc new_loc) = exp_val comp_s new_rel
        new_s = update_store comp_s
                (updateStore (store comp_s) new_loc (Vcomp new_r))

do_establish_rel :: State -> Ide -> CompID -> CompID -> (State,CompID)
do_establish_rel s rk p c = (new_s,new_r) where
        (r_s,new_r) = add_component s rk
        new_rs = updateCompStore (updateCompStore (comps r_s) new_r "parent" (Rv (Vcomp p)))
                new_r "child" (Rv (Vcomp c))
        new_pcr = updateCompStore new_rs p "relationships" (Rv (Vlist (comps_to_values([new_r]++comp_rels r_s
p))))
        new_pcc = updateCompStore new_pcr c "relationships" (Rv (Vlist (comps_to_values ([new_r]++comp_rels r_s
c))))
        updated_s = update_comps r_s new_pcc
        new_s = update_dependents updated_s c
                (UpdateRec "establish_rel" [Vstring rk,Vcomp p,Vcomp c])

-- Convert CompIDs to Values
--
comps_to_values :: [CompID] -> [Value]
comps_to_values [] = []
comps_to_values (c:cs) = [comp_to_value c]++comps_to_values cs

comp_to_value :: CompID -> Value
comp_to_value c = (Vcomp c)

-- establish_rel(in Kind,in Parent,in Child)
--
establish_link_rel :: State -> Command -> State
establish_link_rel s (EstablishLink kind parent child) = new_s where
        rk = rel_kind_type kind
        (Vcomp p) = rval s (exp_val s parent)
        (Vcomp c) = rval s (exp_val s child)
```

```
                (new_s,new_r) = do_establish_rel s rk p c


-- reestablish_rel(in Rel,in Parent,in Child)
--
reestablish_rel :: State -> Command -> State
reestablish_rel s (Reestablish rel parent child) = new_s where
        (Vcomp r) = rval s (exp_val s rel)
        (Vcomp p) = rval s (exp_val s parent)
        (Vcomp c) = rval s (exp_val s child)
        new_s = do_reestablish_rel s r p c


do_reestablish_rel s r p c = new_s where
        (CValue (Rv (Vcomp old_p))) = comps s r "parent"
        (CValue (Rv (Vcomp old_c))) = comps s r "child"
        (CValue (Rv k)) = (comps s) r "class"
        -- "disolve" relationship for old_p/old_c
        updated_s1 = update_dependents s r
                (UpdateRec "disolve_rel" [k,Vcomp old_p,Vcomp old_c])
        dissolved_pcr = updateCompStore (comps updated_s1) old_p "relationships"
                (Rv (Vlist (comps_to_values (remove_all [r] (comp_rels updated_s1 old_p) (==) ))))
        dissolved_pcc = updateCompStore dissolved_pcr old_c "relationships"
                (Rv (Vlist (comps_to_values (remove_all [r] (comp_rels updated_s1 old_c) (==) ))))
        updated_s2 = update_comps updated_s1 dissolved_pcc
        -- "establish" relationship for p/c
        new_rs = updateCompStore (updateCompStore (comps updated_s2) r "parent" (Rv (Vcomp p)))
                r "child" (Rv (Vcomp c))
        new_pcr = updateCompStore new_rs p "relationships"
                (Rv (Vlist (comps_to_values ([r]++comp_rels updated_s2 p))))
        new_pcc = updateCompStore new_pcr c "relationships"
                (Rv (Vlist (comps_to_values ([r]++comp_rels updated_s2 c))))
        updated_s3 = update_comps updated_s2 new_pcc
        new_s = update_dependents updated_s3 c
                (UpdateRec "establish_rel" [k,Vcomp p,Vcomp c])


-- disolve_rel(in Kind,in Parent,in Child)
--
disolve_rel :: State -> Command -> State
disolve_rel s (Disolve kind parent child) = new_s where
        rk = rel_kind_type kind
        vp@(Vcomp p) = rval s (exp_val s parent)
        vc@(Vcomp c) = rval s (exp_val s child)
        rs = filter (rel_kind_child s rk c) (comp_rels s p)
        dislove_rels [] s = s
        disolve_rels (r:rs) s = disolve_rels rs (do_disolve_rel s r)
        new_s = disolve_rels rs s


-- Find all relationships with same kind/child from parent
--
rel_kind_child :: State -> Ide -> CompID -> CompID -> Bool
rel_kind_child s kind child rel = result where
        (CValue (Rv (Vcomp c))) = comps s rel "child"
        (CValue (Rv (Vstring rkind))) = comps s rel "class"
        result = if (child == c) && (rkind==kind) then True else False


-- Find all relationships of given kind i
--
find_kind_rels :: [CompID] -> State -> Ide -> [CompID]
find_kind_rels [] s kind = []
find_kind_rels (r:rs) s kind = rs where
        (CValue (Rv (Vstring rkind))) = (comps s) r "class"
        rs = if rkind == kind then [r]++find_kind_rels rs s kind
                else find_kind_rels rs s kind


-- Remove all values from relationships and dependents list for
-- dissolved relationship and delete the relationship component.
--
do_disolve_rel :: State -> CompID -> State
do_disolve_rel s r = new_s where
        (CValue (Rv (Vcomp p))) = comps s r "parent"
        (CValue (Rv (Vcomp c))) = comps s r "child"
        (CValue (Rv kind_val)) = (comps s) r "class"
        updated_s = update_dependents s r (UpdateRec "disolve_rel" [kind_val,Vcomp p,Vcomp c])
```

```
                dissolved_r = disolve_relationships (comp_rels updated_s r) updated_s
                dissolved_pcr = updateCompStore (comps updated_s) p "relationships"
                        (Rv (Vlist (comps_to_values (remove_all [r] (comp_rels updated_s p) (==) ))))
                dissolved_pcc = updateCompStore dissolved_pcr c "relationships"
                        (Rv (Vlist (comps_to_values (remove_all [r] (comp_rels updated_s c) (==) ))))
                new_s = delete_comp (update_comps updated_s dissolved_pcc)  r

-- create_view(in kind,out ViewID)
--
create_view :: State -> Command -> State
create_view s (CreateView kind new_view) = new_s where
        (comp_s,new_v) = do_create_view s kind
        (Loc new_loc) = exp_val comp_s new_view
        new_s = update_store comp_s (updateStore (store comp_s) new_loc (Vcomp new_v))


do_create_view s kind = (new_s,new_v) where
        (v_s,new_v) = add_component s kind
        new_s = update_comps v_s (updateCompStore (comps v_s) new_v "components" (Rv (Vlist [])))


-- add_view_element(in View,in Comp)
--
add_view_component :: State -> Command -> State
add_view_component s (AddViewComponent view comp) = new_s where
        (Vcomp v) = rval s (exp_val s view)
        (Vcomp c) = rval s (exp_val s comp)
        new_s = do_add_view_component s v c

do_add_view_component s v c = new_s where
        new_comps1 = updateCompStore (comps s) c "view" (Rv (Vcomp v))
        new_comps2 = updateCompStore new_comps1 v "components"
                (Rv (Vlist (comps_to_values (view_comps s v++[c]))))
        new_s = update_comps s new_comps2


-- remove_view_element(in View,in Comp)
--
remove_view_component :: State -> Command -> State
remove_view_component s (RemoveViewComponent view comp) = new_s where
        (Vcomp v) = rval s (exp_val s view)
        (Vcomp c) = rval s (exp_val s comp)
        new_s = do_remove_view_component s v c

do_remove_view_component s v c = new_s where
        new_comps1 = updateCompStore (comps s) c "view" (Rv Nil)
        new_comps2 = updateCompStore new_comps1 v "components"
                (Rv (Vlist (comps_to_values (remove_all [c] (view_comps s v) (==) ))))
        new_s = update_comps s new_comps2


-- Remove component from its view (if its in one)
--
remove_from_view :: State -> CompID -> State
remove_from_view s c = remove_view s c view where
        view = (comps s) c "view"
        remove_view s c NoCValue = s
        remove_view s c (CValue (Rv (Vcomp v))) = do_remove_view_component s v c


-- store_update(in Comp,in UpdateValue)
--
-- Updates are stored in default attribute "updates" for every component
--
store_update :: State -> Command -> State
store_update s (Store expr kind args) = new_s where
        arg_vals [] s = []
        arg_vals (x:xs) s = [rval s (exp_val s x)]++arg_vals xs s
        (Vcomp c) = rval s (exp_val s expr)
        (CValue (Rv (Vlist c_updates))) = (comps s) c "updates"
        new_updates = c_updates++[Vlist ([Vstring kind]++arg_vals args s)]
        -- i.e. updates stored as list of the form: [kind,Value1,...,Valuen]
        new_s = update_comps s (updateCompStore (comps s) c "updates" (Rv (Vlist new_updates)))


-- CompExp.OpName([ArgExp])
--
-- Component-specific operation meaning is:
```

```
--  - compute arguments
--  - allocate component values
--      (c.f. OO language method - scope = object's class values + args & locals)
--  - allocate value/variable arguments
--  - allocate locals
--  - allocate "self" local
--  - get meaning of operation command
--  - deallocate self
--  - deallocate locals
--  - deallocate arguments
--  - deallocate component values
--
apply_operation :: State -> Command -> State
apply_operation s (ApplyOp exp arg_exps) = new_s where
        (CompValue c op) = exp_val s exp
        (CValue (Rv (Vstring ct))) = (comps s c "class")
        (TCompData bk vs cts) = (declarations s) ct
        arg_vals :: [Exp] -> State -> [Value]
        arg_vals [] s = []
        arg_vals (e:es) s = (arg_vals es s)++[(exp_rval s e)]
        new_s = case (cts op) of
                (COperation args t locs command) -> op_result where
                        arg_vals = eval_args arg_exps s
                        old_env = env s
                        pre_op_s = alloc_self (alloc_locals (alloc_and_bind_args
        (alloc_comp_values s vs c) args arg_vals) locs) c
                        post_op_s = dealloc_comp_values (dealloc_args (dealloc_locals
        (dealloc_self (command command_meaning pre_op_s)) locs) args) vs c
                        op_result = update_env post_op_s old_env
                    (CUpdates updates) -> apply_updates updates s c (arg_vals arg_exps s) vs
                        -- call update operation as an operation


-- Evaluate lvalues for arguments
--
eval_args :: [Exp] -> State -> [Dv]
eval_args [] s = []
eval_args (e:es) s = (eval_args es s)++[(exp_val s e)]


-- Allocate component values
--
alloc_comp_values :: State -> [Ide] -> CompID -> State
alloc_comp_values s [] c = s
alloc_comp_values s (n:ns) c = alloc_comp_values new_s ns c where
        new_env = updateEnv (env s) n (CompValue c n)
        new_s = update_env s new_env


-- Allocate & bind arguments for operation
--
-- In arguments have new location which is the Value of actual argument
--   (i.e. value parameters)
-- Out arguments have same Dv as actual argument
--   (i.e. variable parameters)
--
alloc_and_bind_args :: State -> OpArgs -> [Dv] -> State
alloc_and_bind_args s [] [] = s
alloc_and_bind_args s ((n,In,_):as) (v:vs) = new_s where
        rv = rval s v
        l = new (store s)
        new_store = updateStore (store s) l rv
        new_env = updateEnv (env s) n (Loc l)
        new_s = alloc_and_bind_args (update_store (update_env s new_env) new_store) as vs
alloc_and_bind_args s ((n,Out,_):as) (v:vs) = new_s where
        new_env = updateEnv (env s) n v
        new_s = alloc_and_bind_args (update_env s new_env) as vs


-- Allocate locals for operation
--
alloc_locals :: State -> OpLocals -> State
alloc_locals s args = new_s where
        loc_names [] = []
        loc_names ((n,_):ns) = [n]++loc_names ns
        an = loc_names args
```

```
                  (ls,new_store) = news (length an) (store s)
                  new_env = extendEnv an ls (env s)
                  new_s = update_store (update_env s new_env) new_store


-- Allocate "self" variable for operation
--
alloc_self :: State -> CompID -> State
alloc_self s c = new_s where
                  l = new (store s)
                  new_store = updateStore (store s) l (Vcomp c)
                  new_env = updateEnv (env s) "self" (Loc l)
                  new_s = update_store (update_env s new_env) new_store


-- Deallocate a list of identifiers from Store
--
dealloc :: State -> [Ide] -> State
dealloc s ns = new_s where
                  dealloc_ids :: [Ide] -> Env -> Store -> (Env,Store)
                  dealloc_ids [] e s = (e,s)
                  dealloc_ids (n:ns) e s = dealloc_ids ns (deallocEnv e n) (deallocStore s l) where
                            (Bound (Loc l)) = e n
                  (new_env,new_store) = dealloc_ids ns (env s) (store s)
                  new_s = update_store (update_env s new_env) new_store


-- Deallocate "self" variable for operation
--
dealloc_self :: State -> State
dealloc_self s = dealloc s ["self"]


-- Deallocate arguments for operation
--
dealloc_args :: State -> OpArgs -> State
dealloc_args s [] = s
dealloc_args s ((n,In,_):as) = dealloc_args (dealloc s [n]) as
dealloc_args s ((n,Out,_):as) = new_s where
                  new_env = deallocEnv (env s) n
                  new_s = dealloc_args (update_env s new_env) as


-- Deallocate locals for operation
--
dealloc_locals :: State -> OpLocals -> State
dealloc_locals s args = new_s where
                  loc_names [] = []
                  loc_names ((n,_):ns) = [n]++loc_names ns
                  an = loc_names args
                  new_s = dealloc s (loc_names args)


-- Deallocate component values
--
dealloc_comp_values :: State -> [Ide] -> CompID -> State
dealloc_comp_values s [] c = s
dealloc_comp_values s (n:ns) c = dealloc_comp_values new_s ns c where
                  new_s = update_env s (deallocEnv (env s) n)


-- record_update(in Exp, in Kind, in [Exp])
--
record_update :: State -> Command -> State
record_update s (Record comp_exp kind values) = new_s where
                  eval_exps :: [Exp] -> State -> [Value]
                  eval_exps [] s = []
                  eval_exps (e:es) s =
                            [(exp_rval s e)]++eval_exps es s
                  (Vcomp c) = exp_rval s comp_exp
                  new_s = update_dependents s c (UpdateRec kind (eval_exps values s))


-- Dependents for a component are:
--  1) itself
--  2) all relationships it participates in
--  3) all other components its connected to via its relationships
--
dependents :: State -> CompID -> [CompID]
dependents s c = deps where
```

```
              rs = comp_rels s c
              deps = [c]++rs++collect_deps rs s c
              collect_deps [] s c = []
              collect_deps (x:xs) s c = cd where
                      (CValue (Rv (Vcomp parent))) = comps s x "parent"
                      (CValue (Rv (Vcomp child))) = comps s x "child"
                      cd = if parent == c then [child]++collect_deps xs s c
                              else [parent]++collect_deps xs s c


-- Send update record to dependents for a component
--
update_dependents :: State -> CompID -> UpdateRecord -> State
update_dependents s c u = new_s where
        update_dependents1 [] s _ = s
        update_dependents1 (d:ds) s u =
                update_dependents1 ds (update_from s d u) u
        output_s = update_output s ((output s)++[(OV c u)])
        new_s = update_dependents1 (dependents s c) output_s  u


-- Process update from another component
--
update_from :: State -> CompID -> UpdateRecord -> State
update_from s d (UpdateRec kind arg_vals) = new_s where
        (CValue (Rv (Vstring k))) = comps s d "class"
        (TCompData bk vs ct) = (declarations s) k
        new_s = case (ct kind) of
                (CUpdates updates) -> apply_updates updates s d arg_vals vs
                _ -> s


-- Apply an update to a component (if it supports the update)
--
-- Update operations are performed by finding a match (correct kind,
-- number and type of args and guard that evaluates to true) and
-- applying the operation as for component-specific operations
--
apply_updates :: [CUpdate] -> State -> CompID -> [Value] -> [Ide] -> State
apply_updates [] s d arg_vals vs = s
apply_updates ((UpdateOp args locs g command):us) s d arg_vals vs =
        if same_length_and_type (reverse args) arg_vals s
                then upd_s else apply_updates us s d arg_vals vs where
                        vals :: [Value] -> [Dv]
                        vals [] = []
                        vals (v:vs) = (vals vs)++[Rv v]
                        old_env = env s
                        pre_op_s = alloc_self (alloc_locals (alloc_and_bind_args
        (alloc_comp_values s vs d) args (vals arg_vals)) locs) d
                        upd_s = case (g exp_rval pre_op_s) of
                                (Vbool True) -> op_result where
                        post_op_s = dealloc_comp_values (dealloc_self (dealloc_locals (
                                dealloc_args (command command_meaning pre_op_s) args) locs)) vs d
                        op_result = update_env post_op_s old_env
                                _ -> apply_updates us s d arg_vals vs


same_length_and_type :: OpArgs -> [Value] -> State -> Bool
same_length_and_type [] [] s = True
same_length_and_type [] (a:b) s = False
same_length_and_type (a:b) [] s = False
same_length_and_type ((n,io,tv):as) (v:vs) s =
        if tv == (value_to_type v s) || (value_to_type v s) == TAny
                then same_length_and_type as vs s else False


value_to_type :: Value -> State -> TypeValue
value_to_type (Vbool b) s = TBool
value_to_type (Vnum i) s = TInteger
value_to_type (Vstring st) s = TString
value_to_type (Vlist []) s = TList TAny
value_to_type (Vlist (h:t)) s = TList (value_to_type h s)
value_to_type (Vcomp c) s = TComp k where
        (CValue (Rv (Vstring k))) = (comps s c "class")
value_to_type Nil s = TAny
value_to_type _ s = TNotDefined
```

```
--
-- Expression values for MVSL
--

-- Get the value (Value) of an expression (i.e. an rvalue)
--
rval :: State -> Dv -> Value
rval s (Loc l) = r where (Used r) = (store s) l
rval s (Rv v) = v
rval s (CompValue c a) = cv where
        (CValue (Rv (Vstring ct))) = (comps s c "class")
        (TCompData bk vs comp_types) = (declarations s) ct
        cv = case (comp_types a) of
                (CAttribute t) -> av where
                        (CValue (Rv av)) = (comps s c) a
                (CRelationship t) -> (rel_value c s a t)
                (COperation [] t [] command) -> fn_result where
                        old_env = env s
                        pre_op_s = command command_meaning (alloc_self (alloc_comp_values s vs c) c)
                        (Bound result) = (env pre_op_s) "result"
                        fn_result = rval pre_op_s result

-- Get the denotable value for an expression (i.e. an lvalue)
--
-- The value of a functional operation name is one of:
--  - if name is CAttribute, = component attribute value
--  - if name is CRelationship, = one of:
--         - relationship components (where component is parent or child)
--      - component (if relationship is one-to-one link rel)
--      - list of components (if relationship is one-to-many link rel)
--  - if name is COperation = value of function (value of "result" after
--         executing function as per component-specific operations
--
exp_val :: State -> Exp -> Dv
exp_val _ (IntLit i) = Rv (Vnum i)
exp_val _ (StringLit s) = Rv (Vstring s)
exp_val _ True_ = Rv (Vbool True)
exp_val _ False_ = Rv (Vbool False)
exp_val s (Ident i) = ev where
        (Bound ev) = (env s) i
exp_val s (CompVal e a) = (CompValue c a) where
        (Vcomp c) = exp_rval s e
exp_val s (FuncOp c_exp arg_exps) = ev where
        (CompValue c a) = exp_val s c_exp
        (CValue (Rv (Vstring ct))) = (comps s c "class")
        (TCompData bk vs comp_types) = (declarations s) ct
        ev = case (comp_types a) of
                (COperation args t locs command) -> fn_result where
                        arg_vals = eval_args arg_exps s
                        pre_op_s = command command_meaning (alloc_self (alloc_locals
                                (alloc_and_bind_args (alloc_comp_values s vs c) args arg_vals) locs) c)
                        (Bound result) = (env pre_op_s) "result"
                        fn_result = (Rv (rval pre_op_s result))
                _ -> (CompValue c a)
exp_val s (Op op lexpr rexpr) = opval op lv rv where
        lv = rval s (exp_val s lexpr)
        rv = rval s (exp_val s rexpr)
        opval Plus    (Vnum a) (Vnum b) = (Rv (Vnum (a+b)))
        opval Minus   (Vnum a) (Vnum b) = (Rv (Vnum (a-b)))
        opval Times   (Vnum a) (Vnum b) = (Rv (Vnum (a*b)))
        opval Divide  (Vnum a) (Vnum b) = (Rv (Vnum (a/b)))
        opval Gt      (Vnum a) (Vnum b) = (Rv (Vbool (a>b)))
        opval Lt      (Vnum a) (Vnum b) = (Rv (Vbool (a<b)))
        opval Eq      a b = (Rv (Vbool (a==b)))
        opval Neq     a b = (Rv (Vbool (a/=b)))
        opval And     (Vbool a) (Vbool b) = (Rv (Vbool (a&&b)))
        opval Or      (Vbool a) (Vbool b) = (Rv (Vbool (a||b)))
        opval Append  (Vlist a) (Vlist b) = (Rv (Vlist (a++b)))
        opval Remove  (Vlist a) (Vlist b) = (Rv (Vlist (remove_all b a (==))))

-- Get the Value for an expression
--
```

```
exp_rval :: State -> Exp -> Value
exp_rval s e = rval s (exp_val s e)

-- List manipulation functions
--

-- append_once - append new values to list if not already members of list
--
append_once :: [a] -> [a] -> (a->a->Bool) -> [a]
append_once list append compare = new_list where
        to_append = remove_all list append compare
        new_list = list ++ to_append

-- remove_all - remove all values in first list from second list
--
remove_all :: [a] -> [a] -> (a->a->Bool) -> [a]
remove_all r l compare = new_list where
        remove [] a = []
        remove (x:xs) y = if compare x y then remove xs x else [x]++(remove xs y)
        remove_all' [] l = l
        remove_all' (x:xs) l = remove_all' xs (remove l x)
        new_list = remove_all' r l

-- member - is given CompID a member of the CompID list?
--
member :: a -> [a] -> (a->a->Bool) -> Bool
member e es compare = result where
        member_test [] e = False
        member_test (x:xs) y = if compare x y then True else member_test xs y
        result = member_test es e

-- rel_value
--
-- Value of a relationship is one of:
--  list of relationship components (if relationship type is a component)
--  list of components (if relationship type is one-to-one, one-to-many)
--
rel_value :: CompID -> State -> Ide -> TypeValue -> Value
rel_value c s a (TCompAttr kind porc) = (Vlist (map comp_to_value comps)) where
        rels =filter (rel_kind s kind) (comp_rels s c)
        comps = parent_or_child_comps rels porc s c
rel_value c s a (TOneToOne comp) = rel where
        rk = rel_kind_comp s c a
        rels = filter (rel_kind s rk) (comp_rels s c)
        comps = parent_or_child_comps rels "parent" s c
        rel = if comps == []
                then Nil
                else (Vcomp comp) where
                        (comp:rest) = comps
rel_value c s a (TOneToMany comp) = (Vlist (map comp_to_value comps)) where
        rk = rel_kind_comp s c a
        rels = filter (rel_kind s rk) (comp_rels s c)
        comps = parent_or_child_comps rels "parent" s c

-- Find all relationships of same kind and parent/child value
--
rel_kind :: State -> Ide -> CompID -> Bool
rel_kind s kind r = result where
        (CValue (Rv (Vstring rk))) = comps s r "class"
        result = if (kind==rk) then True else False
rel_kind_and_porc s kind r = result where
        (CValue (Rv (Vstring rk))) = comps s r "class"
        result = if (kind==rk) then True else False

-- Construct list of connected components (parent/child of relationships)
--
parent_or_child_comps :: [CompID] -> Ide -> State -> CompID -> [CompID]
parent_or_child_comps [] porc s comp = []
parent_or_child_comps (r:rs) "parent" s comp = porc_comps where
        (CValue (Rv (Vcomp p))) = comps s r "parent"
        (CValue (Rv (Vcomp c))) = comps s r "child"
        porc_comps = if p == comp then [c]++parent_or_child_comps rs "parent" s comp else
```

```
                          parent_or_child_comps rs "parent" s comp
parent_or_child_comps (r:rs) "child" s comp = porc_comps where
          (CValue (Rv (Vcomp p))) = comps s r "parent"
          (CValue (Rv (Vcomp c))) = comps s r "child"
          porc_comps = if c == comp then [p]++parent_or_child_comps rs "child" s comp
                                    else parent_or_child_comps rs "child" s comp


-- "kind" for a component relationship given a component/attribute name pair
--
rel_kind_comp :: State -> CompID -> Ide -> Ide
rel_kind_comp s c a = k++"."++a where
          (CValue (Rv (Vstring k))) = comps s c "class"


-- "kind" for a relationship given a Type
-- The kind is either the relationship's component name or the owner/relationship name
--
rel_kind_type :: Type -> Ide
rel_kind_type (ComponentType c) = c
rel_kind_type (CompAttrType c a) = c++"."++a



--
-- MVSL program meaning
--
-- Given a type map and a sequence of "inputs", produce a sequence of "outputs"
--
-- Inputs are updates generated by MVisual (i.e. update records) and are translated into
-- operations by the current view.
--
-- Outputs are updates produced by executing MVSL operations on the current state
-- (which also stores the current inputs and outputs).
--

data Input = IV Ide [Value]

-- Meaning of a Program is defined by its outputs given a set of inputs and definition
--
program :: Program -> [Input] -> [Output]
program (Pro decls command) i = out where
          (dv,gs) = program_decls decls emptyDeclValue []
          init_s = alloc_globals (emptyState dv) gs
          com_s = command_meaning init_s command
          out = output (run_program i com_s)

-- Need globals for program definition
--
alloc_globals :: State -> [Ide] -> State
alloc_globals s gs = new_s where
          (ls,new_store) = news (length gs) (store s)
          new_env = extendEnv gs ls (env s)
          new_s = update_store (update_env s new_env) new_store

-- Program is "run" by interpreting a sequence of "inputs" from MVisual
--
run_program :: [Input] -> State -> State
run_program [] s = s
run_program (i:is) s = new_s where
          new_s = run_program is (apply_input_update i s)

-- Translate input "update" record into operation on a component
--
-- Conceptually, MVisual generates these updates in respose to user interaction
-- MVSL's outputs are interpreted by MVisual which then updates view renderings
-- to indicate program change
--
apply_input_update :: Input -> State -> State
apply_input_update (IV "update_attribute" [Vcomp c,Vstring name,new]) s =
          assign_result s (CompValue c name) new
apply_input_update (IV "add_element" [Vstring kind]) s = new_s where
          (new_s,_) = add_component s kind
apply_input_update (IV "delete_component" [Vcomp c]) s =
          do_delete_component s c
```

```
apply_input_update (IV "establish_rel" [Vstring kind,Vcomp parent,Vcomp child]) s = new_s where
        (new_s,_) = do_establish_rel s kind parent child
apply_input_update (IV "reestablish_rel" [Vcomp r,Vcomp p,Vcomp c]) s =
        do_reestablish_rel s r p c
apply_input_update (IV "disolve_rel" [Vcomp r]) s =
        do_disolve_rel s r
apply_input_update (IV "create_view" [Vstring kind]) s = new_s where
        (new_s,_) = do_create_view s kind
apply_input_update (IV "add_view_component" [Vcomp v,Vcomp e]) s =
        do_add_view_component s v e
apply_input_update (IV "remove_view_component" [Vcomp v,Vcomp e]) s =
        do_remove_view_component s v e
apply_input_update (IV "update" [Vcomp c,Vlist (Vstring kind:upd)]) s =
        update_from s c (UpdateRec kind upd)
apply_input_update _ s = s
```

Appendix D

An MVSL Specification of IspelM

```
-- Global values
--
program : program
    -- base view reference

-- Initial computation
--
initialise
    add_element(program,program)
    record_update(program,"init",[])
end initial

-- Program for IspelM
--
base view program
    attributes
        name : string

    relationships
        clusters : one-to-many cluster
        classes : one-to-many class

    operations
        -- Locate a class...
        --
        find_class(in name : like class.name) : class is
        local
            aclass : class
        begin
            result := nil
            forall aclass on classes do
                if aclass.name = name then
                    result := aclass
                end if
            end forall
        end find_class

    updates
        -- Initialise program details
        --
        -- This update is send by the program_details dialog
        --
        details(in pname:string, in clname:string, in cname:string,
            in ckind:like base_class.kind) local
            cluster : cluster
            class : class
            view : class_diagram
            icon : class_icon
        is
            program.name:=pname
            add_element(cluster,cluster)
            cluster.name:=clname
            establish(program.clusters,program,cluster)
            cluster.add_class(cname,ckind,class)
            create_view(class_diagram,view)
            add_element(class_icon,icon)
```

```
                add_view_element(view,icon)
                view.name:='root class'
                establish(icon.base,class,icon)
            end details

end program

-- Base cluster element
--
base element cluster
    attributes
        cluster_name : string

    relationships
        classes : one-to-many class

    operations
        -- class manipulation
        --
        add_class(in name : like class.class_name, in kind : like class.kind,
            out new_class : class) is
            add_element(class,new_class)
            new_class.kind := kind
            establish(cluster.classes,self,new_class)
            establish(program.classes,program,new_class)
        end add_class

        remove_class(in class : class) is
            dissolve(cluster.classes,self,class)
            dissolve(program.classes,program,class)
        end remove_class

end cluster

-- The base element class
--
base element class
    attributes
        class_name : string
        kind : [normal, abstract]

    relationships
        cluster : one-to-one cluster
        generalisations : generalisation.child
        client_suppliers : client_supplier.parent
        classifiers : classifier.parent
        features : one-to-many feature
        specialisations : one-to-many class
        all_features : one-to-many all_feature

    operations
        -- add/remove/find feature
        --
        add_feature(in name : like feature.feature_name,
            in kind : like feature.kind,
            in type : like feature.type_name,
            out new_feature : feature)
        is
            add_element(feature,new_feature)
            new_feature.init(kind,type)
            establish(class.features,self,new_feature)
        end add_feature

        remove_feature(in name : like feature.feature_name) local
            feature : feature
        is
            forall feature on features do
                if feature.feature_name = name then
                    delete_comp(feature)
                end if
            end forall
        end remove_feature
```

```
find_feature(in name : like feature.feature_name) : feature local
    feature : feature
is
    result := nil
    forall feature on features
        if feature.name = name then
            result := feature
        end if
    end forall
end find_feature


-- add/remove generalisations
--
add_gen(in parent : class, out new_rel : generalisation) is
    establish(generalisation,parent,self,new_rel)
end add_gen

remove_gen(in parent : class) is
    dissolve(generalisation,parent,self)
end remove_gen


-- add/remove/find client-suppliers
--
add_cs(
    in ckind : like client_supplier.kind,
    in cfeature : like client_supplier.client_feature,
    in cname : like client_supplier.client_name,
    in stype : like client_supplier.supplier_type,
    in sfeature : like client_supplier.supplier_feature,
    out new_cs : client_supplier)
is
    if program.find_class(stype) then
        establish(client_supplier,self,program.find_class(stype),new_cs)
    else
        establish(client_supplier,self,nil,new_cs)
    end if
    new_cs.init(ckind,cfeature,cname,stype,sfeature)
end add_cs

remove_cs(in supplier : class) is
    dissolve(client_supplier,self,supplier)
end remove_cs

find_cs(
    in ckind : like client_supplier.kind,
    in cfeature : like client_feature,
    in cname : like client_name,
    in stype : like supplier_type,
    in sfeature : like supplier_feature) : client_supplier
local
    cs : client_supplier
is
    result := nil
    forall cs on client_suppliers do
        if cs.ckind = ckind and
            cs.client_feature = cfeature and
            cs.client_name = cname and
            cs.stype = stype and
            cs.sfeature = sfeature then
            result := cs
        end if
    end forall
end find_cs

updates
    -- Check rename of class is valid
    --
    update_attribute(class : class,
        name : attribute,
        oldname : string,
        newname : string)
```

```
            where
                class = self and
                name = "class_name"
            is
                if program.find_class(newname) \= self then
                    store_update(self,"error",[base_class_name(new_name)])
                else
                    store_update(self,"rename_class",[oldname,newname])
                end if
            end update_attribute

end class


-- Class interface
--
component all_feature
    attributes
        owning_class : like class.class_name
        owner_feature : like feature.feature_name
        class_name : like feature.feature_name
        kind : like feature.kind
        type : like feature.type_name
end all_feature


-- Base generalisation relationship
--
base relationship generalisation
    parent class
    child class
    relationships
        renames : one-to-many rename

    updates
        -- Updates go to owning_class - not stored by generalisation
        --
        establish(kind : string, parent : generalisation, child : rename)
        where
            kind = "rename" and parent = self
        is
            store_update(parent,"add rename",[self,rename])
        end establish

        dissolve(kind : string, parent : generalisation, child : rename)
        where
            kind = "rename" and parent = self
        is
            store_update(parent,"remove rename",[self,rename])
        end establish

        -- When establish/dissolve generalistions,
        -- maintain specialisations list attribute
        --
        establish(rel:relationship,
            kind : string,
            parent : class,
            child : class)
        where
            rel = self and kind = "generalisation"
        is
            store_update(parent,"add_gen",[child,parent])
            establish(class.specialisations,parent,self)
        end establish

        dissolve(rel:relationship,
            kind:string,
            parent : class,
            child : class)
        where
            rel = self and kind = "generalisation"
        is
            store_update(parent,"remove_gen",[child,parent])
          dissolve(class.specialisations,parent,self)
```

```
                end establish

end generalisation


-- Generalisation renamed features
--
component rename
    attributes
        parent_name : like feature.feature_name
        child_name : like feature.feature_name
end rename


-- Base client-supplier relationship
--
base relationship client_supplier
    parent class
    child class
    attributes
        kind : [aggregate,local,call]
        client_feature : like feature.feature_name
        client_name : like feature.feature_name
        supplier_type : like class.class_name
        supplier_feature : like feature.feature_name

    operations
        create_cs(
                in cfeature : like client_feature,
                in cname : like client_name,
                in stype : like supplier_type,
                in sfeature : like supplier_feature) is
            client_feature := cfeature
            client_name := cname
            supplier_type := stype
            supplier_feature := sfeature
        end create_cs

        -- Compute supplier for client-supplier (from supplier_type value)
        --
        compute_supplier is
            if program.find_class(supplier_type) then
                reestablish(self,parent,program.find_class(supplier_type))
            else
                reestablish(self,parent,nil)
                end if
        end compute_cs

    updates
        -- Recompute supplier on type change
        --
        update_attribute(cs : client_supplier,
            name:string,
            oldtype : like client_supplier.supplier_type,
            newtype : like client_supplier.supplier_type)
        where
            cs = self and name = "supplier_type"
        is
            compute_supplier
            store_update(parent,"update_attribute",
                [self,supplier_type,oldtype,newtype])
        end update_attribute

        -- updates to parent
        --
        update_attribute(cs : client_supplier,
            name : attribute,
            oldtype : string,
            newtype : string) where
            cs = self
        is
            store_update(parent,"update_attribute",[self,name,oldtype,newtype])
        end update_attribute
```

```
    update_attribute(cs : client_supplier,
        name : attribute,
        oldtype : like client_supplier.kind,
        newtype : like client_supplier.kind) where
        cs = self and name = "kind"
    is
        store_update(parent,"change cs kind",[self,"kind",oldtype,newtype])
    end update_attribute

    -- Convert updates for class and store
    --
    establish(rel:relationship,
        kind:string,
        client : class,
        supplier : class) where
        rel = self and kind = "client_supplier"
    is
        store_update(parent,"add_cs",
            [self,client_feature,client_name,supplier_type,supplier_feature])
    end establish

    dissolve(rel:relationship,
        kind:string,
        client : class,
        supplier : class) where
        rel = self and kind = "client_supplier"
    is
        store_update(parent,"remove_cs",
            [self,client_feature,client_name,supplier_type,supplier_feature])
    end establish

end client_supplier

-- Base classifier relationship
--
base relationship classifier
    parent class
    child class
    attributes
        name : string

    updates
        -- Updates to class
        --
        update_attribute(cl : classifier,
            name:attribute,
            oldvalue:string,
            newvalue:string)
        where
            cl = self and name = "name"
        is
            store_update(parent,"rename classifier",
                [self,name,oldvalue,newvalue])
        end update_attribute

        establish(rel:relationship,
            kind : string,
            owner : class,
            classify_to : class)
        where
            rel=self and kind = "classifier"
        is
            store_update(parent,"add_classifier",[self,classify_to])
        end establish

        dissolve(rel:relationship,
            kind : string,
            owner : class,
            classify_to : class)
        where
            rel=self and kind = "classifier"
        is
```

```
            store_update(parent,"remove_classifier",[self,classify_to])
        end establish

end classifier

-- Base feature element
--
base element feature
    attributes
        feature_name : string
        kind : [attribute, method, deferred, inherited]
        type_name : string

    relationships
        owning_class : one-to-one class

    operations
        -- Initialise feature
        --
        init(in new_kind : like feature.kind,
            in new_type : like feature.type_name) is
            kind := new_kind
            type_name := new_type
        end init

    updates
        -- Updates against feature (if method) and owning_class
        --
        update_attribute(feature:feature,
            name:string,
            old : string,
            new : string)
        where
            feature = self
        is
            if name = "feature_name" then
                store_update(owning_class,"rename feature",
                    [self,old,new])
                if kind = method then
                    store_update(self,"rename feature",[self,old,new])
                end if
            else
                store_update(owning_class,"change feature type",
                [self,old,new])
                if kind = method then
                    store_update(self,"change feature type",
                        ["change feature type",self,old,new])
                end if
            end if
        end update_attribute

end feature

-- Class icons represent class name/kind and arbitrary features (as their names)
--
subset element class_icon
    attributes
        class_name : like class.class_name
        kind : like class.kind
        feature_names : list like feature.feature_name

    relationships
        view : one-to-one class_diagram_view
        base : one-to-one class

    operations
        -- feature name maintenance
        --
        add_feature_name(in name : like feature.feature_name) is
            feature_names := feature_names ++ {name}
        end add_feature
```

```
        remove_feature_name(in name : like feature.feature_name) is
            feature_names := feature_names -- {name}
        end remove_feature

        -- reselect new class
        --
        reselect_class(in name : like class.class_name)
        local
            other_class : class
        is
            other_class := program.find_class(name)
            if other_class \== self then
                dissolve(base,base,self)
                class_name := name
                map
            end if
        end reselect_class

        -- Map this class icon to a base class
        --
        map(in do_map : boolean)
            base_class : class
        is
            base_class := program.find_class(class_name)
            if base_class \== nil then
                if do_map then
                    establish(base,base_class,self)
                end if
            else
                if do_map then
                    program.default_cluster.add_class(class_name,kind,base_class)
                    establish(sbase,base_class,self)
                end if
            end if
        end map

    updates
        -- Change/Remap a feature name
        --
        change_feature(name : like feature.feature_name,
            new_name:like feature.feature_name,
            new_type:like feature.type_name,
            new_kind:like feature.kind,
            show:boolean)
        local
            feature : base_feature
        is
            if base \== nil then
                feature := base.find_feature(name)
                if feature \== nil then
                    feature.feature_name := new_name
                    feature.type_name := new_type
                    feature.kind := new_kind
                end if
            end if
            remove_feature_name(name)
            if show then
                add_feature_name(new_name)
            end if
        end change_feature

        remap_feature(name, new_name:like feature.feature_name,
            new_type:like feature.type_name,
            new_kind:like feature.kind,
            show:boolean) where true local
            feature : feature
        is
            feature := base.find_feature(new_name)
            if feature = nil then
                base.add_feature(new_name,new_type,new_kind,feature)
            end if
            remove_feature_name(name)
```

```
        if show then
            add_feature_name(new_name)
        end if
    end remap_feature


    -- Translate base feature updates into feature_names
    --
    establish(class : class,
        kind : string,
        class : class,
        feature : feature)
    where
        kind = "class.features"
    is
        add_feature_name(feature.feature_name)
    end establish

    dissolve(class : class,
        kind : string,
        class : class,
        feature : feature)
    where
        kind = "class.features"
    is
        remove_feature_name(feature.feature_name)
    end dissolve


    -- Translate base attribute updates into subset changes
    --
    update_attribute(class : class,
        name : string,
        old : string,
        new : string)
    where
        class = base and name = "class_name"
    is
        class_name := new
    end update_attribute

    update_attribute(class : class,
        name : string,
        old : like class.kind,
        new : like class.kind)
    where
        class = base and name = "kind"
    is
        kind := new
    end update_attribute


    -- Translate subest updates into base updates
    --
    update_attribute(class : class_icon,
        name : string,
        old : string,
        new : string)
    where
        class = self and name = "class_name"
    is
        if base \== nil then
            base.class_name := new
        end if
    end update_attribute

    update_attribute(class : class_icon,
        name : string,
        old : like class.kind,
        new : like class.kind)
    where
        class = self and name = "kind"
    is
        if base \== nil then
            base.kind := new
```

```
            end if
        end update_attribute

end class_icon

-- Generalisation Glue
--
subset relationship generalisation_glue
    parent class_icon
    child class_icon

    relationships
        view : one-to-one class_diagram_view
        base : one-to-one generalisation

    operations
        -- map gen glue to base generalisation
        --
        map(do_map : boolean)
        local
            parent_class : class
            child_class : class
            gen : generalisation
        is
            parent_class := parent.base
            child_class := child.base
            if parent_class \== nil and child_class \== nil then
                gen := child_class.find_gen(parent_class)
                if gen \== nil then
                    if do_map then
                        establish(base,gen,self)
                    end if
                else
                    if do_map then
                        child_class.add_gen(parent_class,gen)
                        establish(base,gen,self)
                    end if
                end if
            end if
        end map

end generalisation_glue

-- Classifier glue
--
subset relationship classifier_glue
    parent class_icon
    child class_icon
    attributes
        name : string

    relationships
        view : one-to-one class_diagram_view
        base : one-to-one classifier

    operations
        -- Map to base classifier
        --
        map(in do_map : boolean)
        local
            parent : class
            child : class
            classifier : classifier
        is
            parent := parent.base
            child := child.base
            if parent \== nil and child \== nil then
                classifier := parent.find_cl(name,child)
                if classifier \== nil then
                    if do_map then
                        establish(base,classifier,self)
                    end if
```

```
                else
                    if do_map then
                        establish(classifier,parent,child,classifier)
                        classifier.init(name)
                        establish(base,classifier,self)
                        result := classifier
                    end if
                end if
            end if
        end map

    updates
        -- Base->subset
        update_attribute(cl : classifier,
            name : string,
            old : string,
            new : string) where
            cl = base and name = "name"
        is
            name := name
        end update_attribute

        -- subset->base
        update_attribute(cl : classifier_glue,
            name : string,
            old : string,
            new : string) where
            cl = self and name = "name"
        is
            if base \== nil then
                base.name := name
            end if
        end update_attribute

end classifier_glue

-- Feature or client-supplier glue
-- (represents features as aggregate client-suppliers)
--
subset relationship cs_or_feature
    parent class_icon
    child class_icon
    attributes
        client_feature : like client_supplier.client_feature
        client_name : like client_supplier.client_name
        supplier_type : like client_supplier.supplier_type
        supplier_feature : like client_supplier.supplier_feature
        kind : like client_supplier.kind

    relationships
        view : one-to-one class_diagram_view
        cs : one-to-one client_supplier
        feature : one-to-one feature

    operations
        -- map to cs or feature
        --
        map(in do_map)
        local
            parent : class
            child : class
            feature : feature
            cs : client_supplier
        is
            parent := parent.base
            child := child.base
            if kind = aggregate then
                -- map to base feature...
                if cs \== nil then
                    disolve(cs,cs,self)
                end if
                if feature = nil and parent \== nil then
```

```
                    feature := parent.find_feature(client_feature)
                    if feature \== nil then
                        if do_map then
                            establish(feature,feature,self)
                        end if
                    else
                        if do_map then
                            parent.add_feature(
                                client_feature,aggregate,supplier_type,feature)
                            establish(feature,feature,self)
                        end if
                    end if
                end if
            else
                if feature \== nil then
                    disolve(feature,feature,self)
                end if
                if cs = nil and parent \== nil then
                    cs := parent.find_cs(kind,client_feature,
                        client_name,supplier_type,supplier_feature)
                    if cs \== nil then
                        if do_map then
                            establish(cs,cs,self)
                        end if
                    else
                        if do_map then
                            parent.add_cs(kind,client_feature,
                                client_name,supplier_type,supplier_feature,cs)
                            establish(cs,cs,self)
                        end if
                    end if
                end if
            end if
        end if
    end susbet_kind

updates
    -- Updates from base feature
    --
    update_attribute(feature : feature,
        name : string,
        old : string,
        new : string)
    where true
    is
        if name = "feature_name" then
            client_feature := new
        else
            if name = "type_name" then
                supplier_type := new
            end if
        end if
    end update_attribute

    -- Updates from client_supplier
    --
    update_attribute(cs : client_supplier,
        name : string,
        old : string,
        new : string)
    where true
    is
        if name = "client_feature" then
            client_feature := new
        else
            if name = "client_name" then
                client_name := new
            else
                if name = "supplier_type" then
                    supplier_type := new
                else
                    if name = "supplier_feature" then
                        supplier_feature := new
```

```
                    end if
                end if
            end if
        end if
    end update_attribute

    -- If supplier_type changed => reselect child class
    --
    update_attribute(cs : cs_or_feature,
        name : string,
        old : string,
        new : string)
    where
        cs = self and name = "supplier_type"
    is
        child.reselect_class(new)
        if feature \== nil then
            feature.type_name := new
            feature.type_class := child
        else
            cs.supplier_type := new
            cs.supplier := child
        end if
    end update_attribute

    -- Updates to base
    --
    update_attribute(cs : cs_or_feature,
        name : string,
        old : string,
        new : string)
    where
        cs = self
    is
        if feature \== nil then
            if name = "client_feature" then
                feature.client_feature := name
            end if
        else
            if name = "client_feature" then
                cs.client_feature := new
            else
                if name = "client_name" then
                    cs.client_name := new
                else
                    if name = "supplier_feature" then
                        cs.supplier_feature := new
                    end if
                end if
            end if
        end if
    end update_attribute

end cs_or_feature

-- Class text
--
subset element class_text
    attributes
        class_name : like base_class.name

    relationships
        view : one-to-one class_code_view
        base : one-to-one class

    updates
        -- parse updates
        parsed_attribute(name : like feature.feature_name,
            type : like feature.type_name) where true local
            feature : feature
        is
            feature := base.find_feature(name)
```

```
            if feature \== nil then
                if feature.type_name \== type then
                    feature.type_name := type
                end if
                feature.kind := attribute
            else
                base.add_feature(name,attribute,type,feature)
            end if
        end parsed_attribute

end class_text

-- Feature text
--
subset element feature_text
    attributes
        class_name : like base_class.class_name
        feature_name : like base_feature.feature_name

    relationships
        view : one-to-one class_code_view
        base : one-to-one class

end feature_text

-- Class diagram view
--
subset view class_diagram_view
    components
        class_icon, generalisation_glue, cs_or_feature, classifier_glue

    attributes
        name : string

    relationships
        focus : one-to-one class

    updates
        -- Expand
        --
        expand(icon : class_icon,
            kind : string,
            gen : generalisation)
        where
            icon.base = gen.parent
        local
            new_icon : class_icon
            new_glue : generalisation_glue
        is
            add_element(class_icon,new_icon)
            add_view_component(self,new_icon)
            establish(generalisation_glue,icon,new_icon,new_glue)
            add_view_component(self,new_glue)
        end expand

        -- add_icon
        --
        add_icon(kind:string,X:integer,Y:integer)
        where
            kind = "class_icon"
        local
            new_class : class_icon
        is
            add_element(class_icon,new_class)
            new_class.x:=X
            new_class.y:=Y
            add_view_element(self,new_class)
            record_update(new_class,"init_details",[])
            new_class.map
        end add_icon

        -- add glue
```

```
            --
            add_glue(kind:string,parent : class_icon,child : class_icon) where
                kind = "generalisation_glue"
            local
                new_gen : generalisation_glue
            is
                establish(generalisation_glue,parent,child,new_gen)
                record_update(new_gen,"init_details",[])
                new_gen.map
            end add_glue

end class_diagram_view

-- Class text view
--
subset view class_code_view
    components
        class_text, feature_text

    attributes
        name : string

    relationships
        class_focus : one-to-one class_text
        feature_focus : one-to-one feature_text

end class_code_view
```

Appendix E

An MVisual Specification for IspelM

This appendix gives a more complete description of the user interaction aspects of IspelM using the MVisual notation introduced in Chapter 5.

E.1. IspelM Component Appearance

Fig E.1. shows the MVisual appearance definition for a `class_icon`. Class icons display the `class_name`, `feature_names` and kind from an MVSL `class_icon`. The border around a class icon is dependent on the value of `kind` and the size of the border must enclose all feature names and class name for the icon.
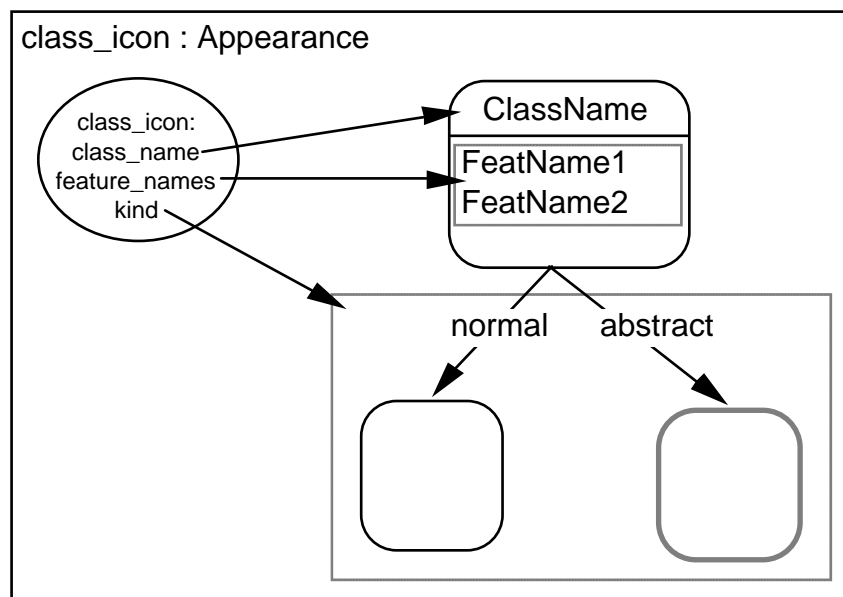
fig E.1.   Class icon appearance.

Fig E.2. shows the appearance of generalisation and classifier glue. Fig E.3. shows the appearance of client-supplier glue. Client-supplier glue comes in four basic appearances:

- code-level aggregates (which are the same as class features) which have a feature (attribute) name
- "local" class references (i.e. method arguments and local variables) which have a feature (method) name and local variable name

- feature calls (method call or attribute fetch) which have an optional client feature name (caller) and optional supplier feature name (called)
- design-level aggregates and locals with no client feature/name values (but which are directional class associations).
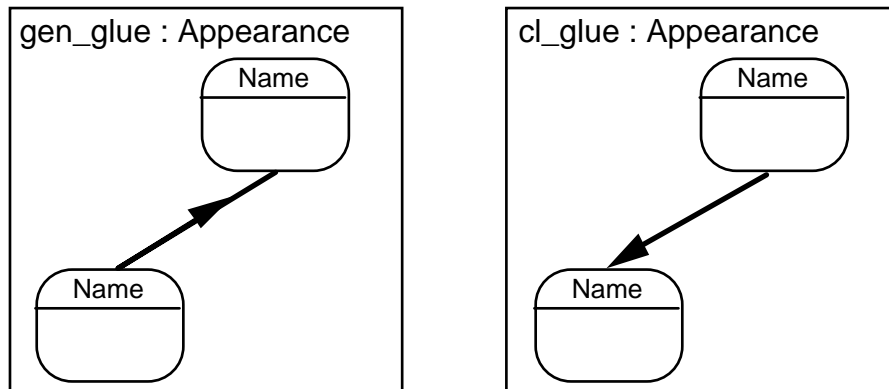

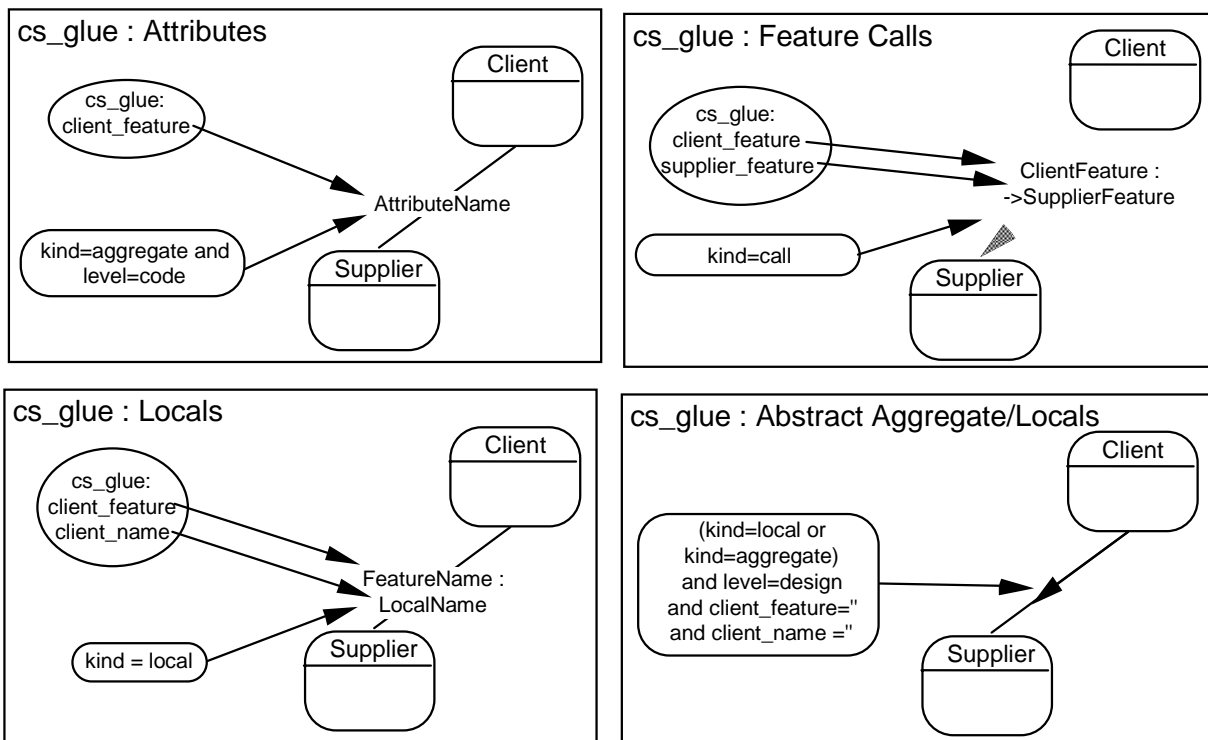fig E.2.   Generalisation and classifier glue.

fig E.3.   Client-supplier glue appearance.

E.2.   IspelM Component Interaction and Updates

Fig E.4. shows the appearance of graphical class diagram views and the additon tools supported by these views. Class diagram views also support tools for arbitrary graphical manipulation (dragging, selecting, clicking, hiding and creating new component views). They provide addition tools for icon, glue and feature additions. The affect of applying these addition tools is to either open dialogs (MVisual views) for the affected graphical entities the tool is applied to or to send updates to MVSL components.
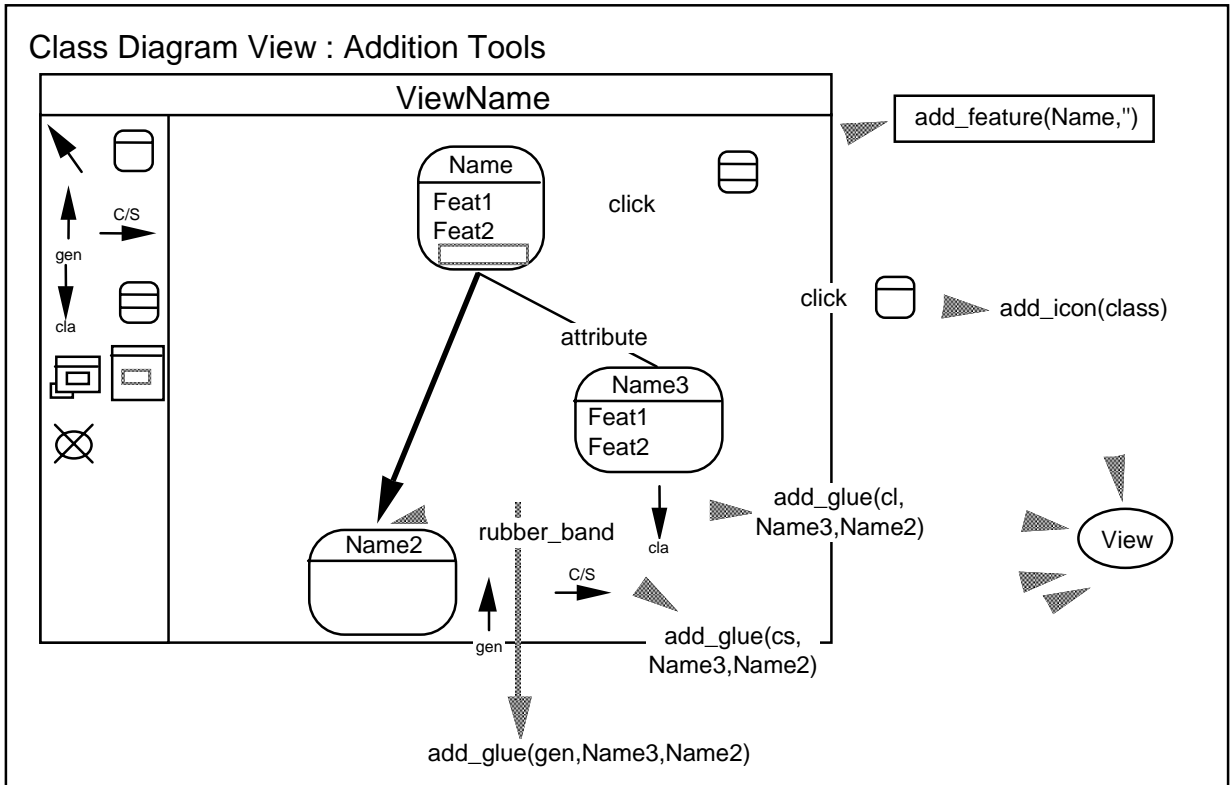
fig. E.4. Appearance of class diagram views and their addition tools.

Existing graphical entities can have their details updated by applying update details tools.
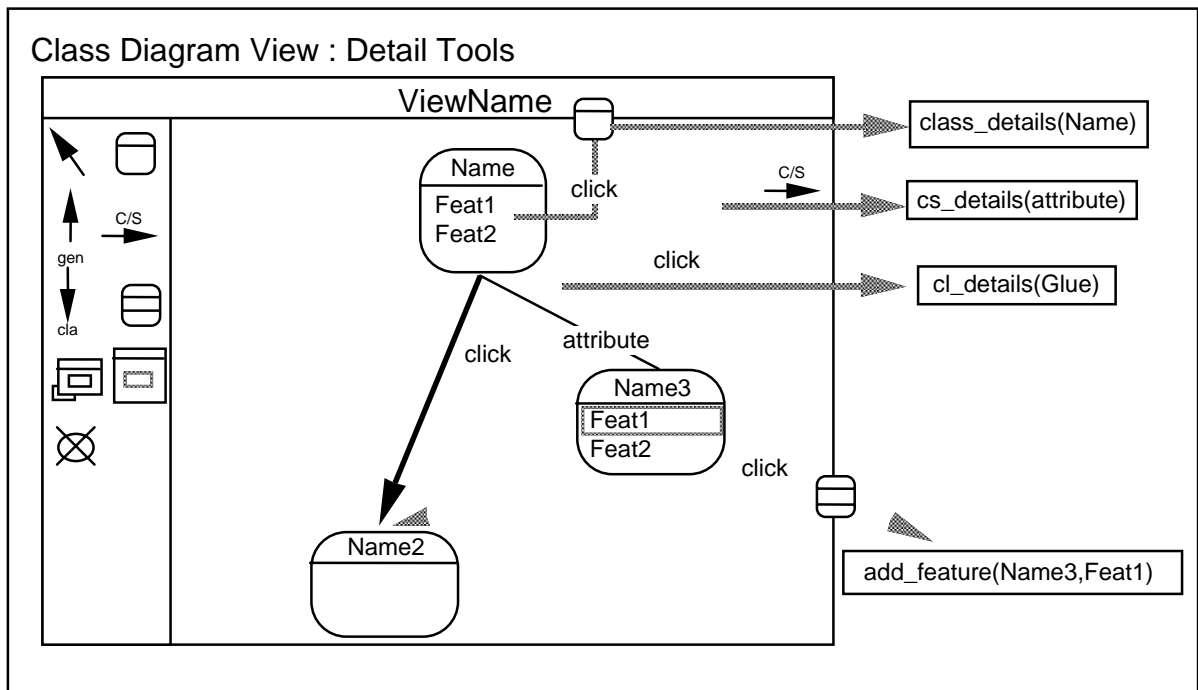Fig. E.5. shows these tools being applied to existing class diagram view entities.

fig. E.5.   Class diagram view update details tools.

Subset class diagram views can expand subset components into a graphical class diagram view. The effect of this expansion is defined in fig. E.6. where different values for an expand update produce different icons and glue.
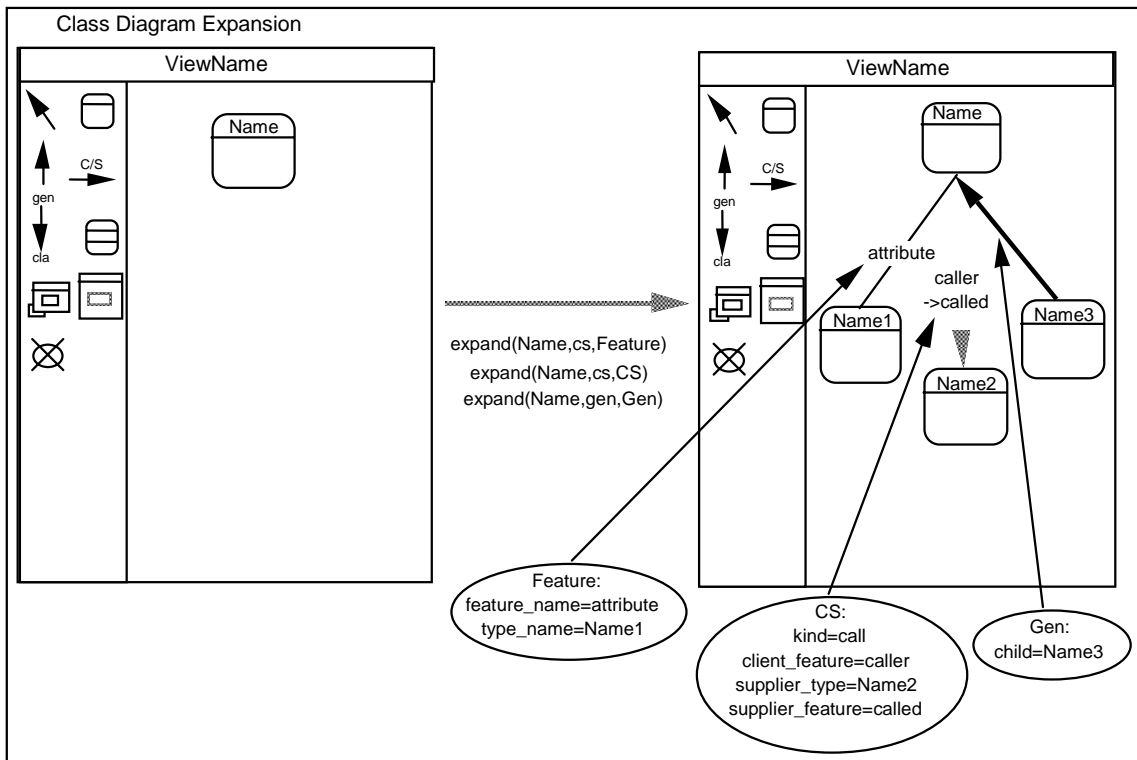
fig E.7.   Class diagram view expansion of icons and glue.

Fig E.8. shows the affect of double-clicking and option-clicking on a class icon. Different parts of the icon produce different affects (the "click point" notion) which is indicated by different updates being sent to views or MVSL components.
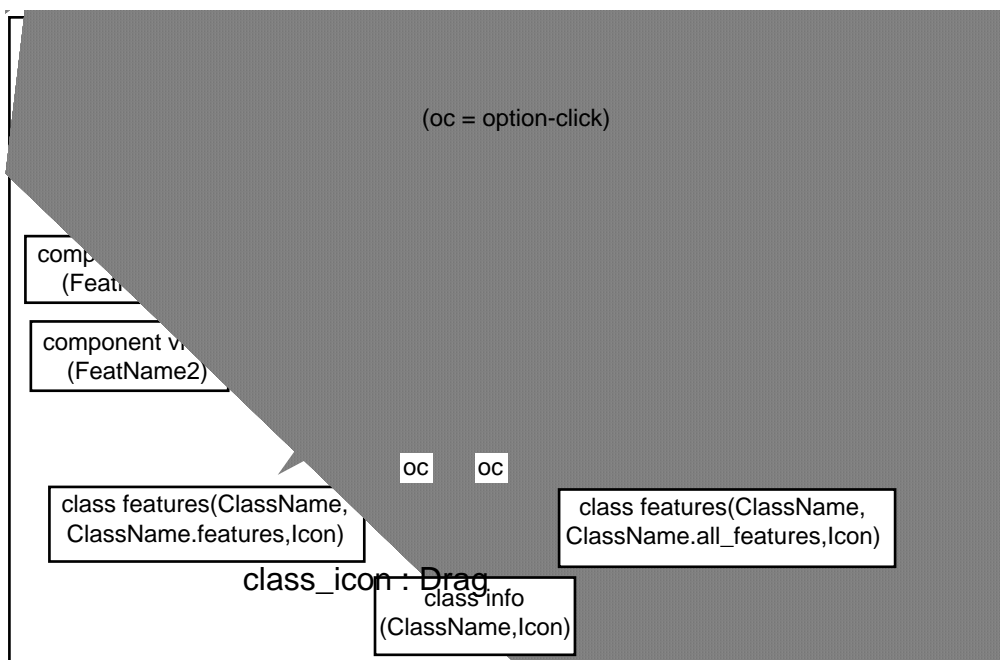

fig E.8.   Double-click/option-click actions for a `class_icon`.

Figs. E.9. (a) and (b) illustrate the affect on a `class_icon` of dragging the icon and renaming the icon name or features it contains. Glue connected to the icon must be redrawn if the icon is moved while the icon must be redrawn to enclose the longest name it contains.
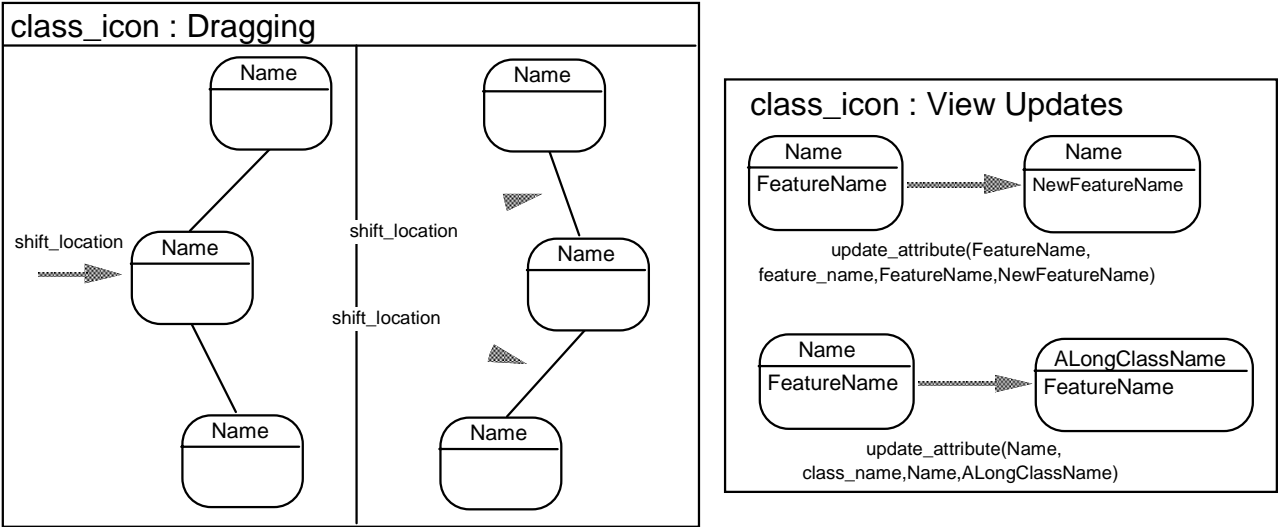

fig E.9. (a) and (b)   Affect on `class_icon` glue when icon dragged and on icon when names are changed.

Fig E.10. illustrates the effect of parsing a textual view (which has a Snart syntax). Parse updates are sent to appropriate MVSL textual forms in the view which will in turn determine if they should send updates to the base view.
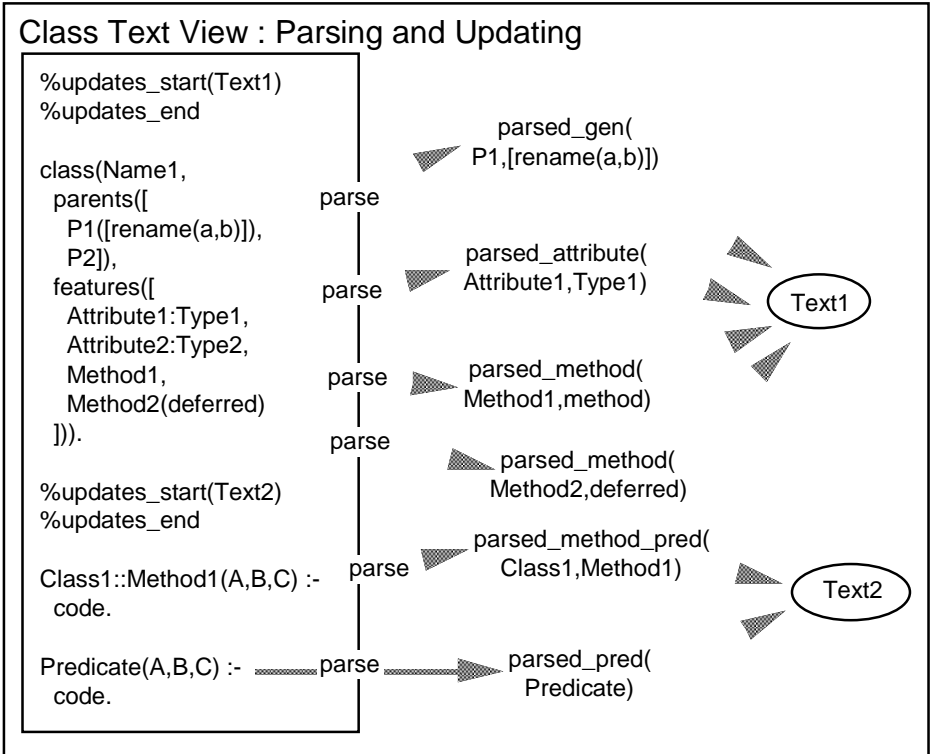

fig E.10. Parsing a textual view with a Snart-like syntax.

Fig E.11. illustrates the effect of unparsing stored base element updates for a dialog or textual view.

```
Class Text View : Stored Base Update Unparsing

rename_class(Old,New) -> "rename class ",Old," to ",New
change_kind(ClassName,Old,New) -> "change class ",ClassName," kind to ",New
add_feature(FeatureName,TypeName) -> "add feature ",FeatureName,":",TypeName
remove_gen(ParentName) -> "remove generalisation to ",ParentName
...
```
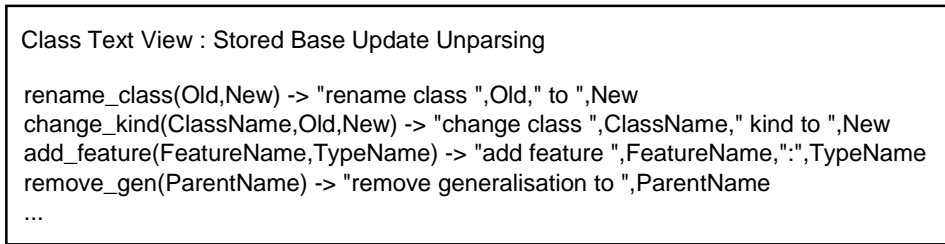
Fig E.11. Unparsing stored updates for textual views or dialogs.

Fig E.12. shows the affect of applying stored updates to a textual view using a Snart-like syntax.
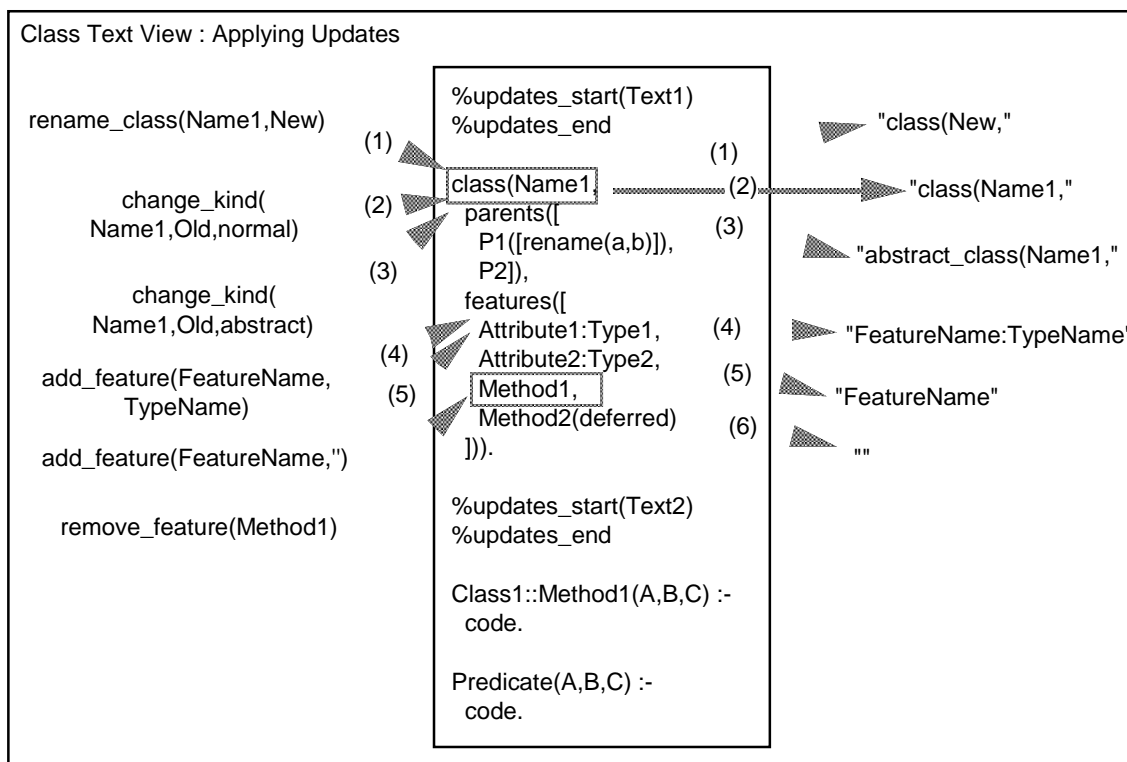

Fig E.12. Affect of applying updates to a textual view with a Snart-like syntax.

E.3.   MViews and IspelM Dialogs

Fig E.13. shows an MViews dialog for selecting a component view from a list of views the component appears in. The affect of selecting the view name is to have the appropriate view sent a `display` update.
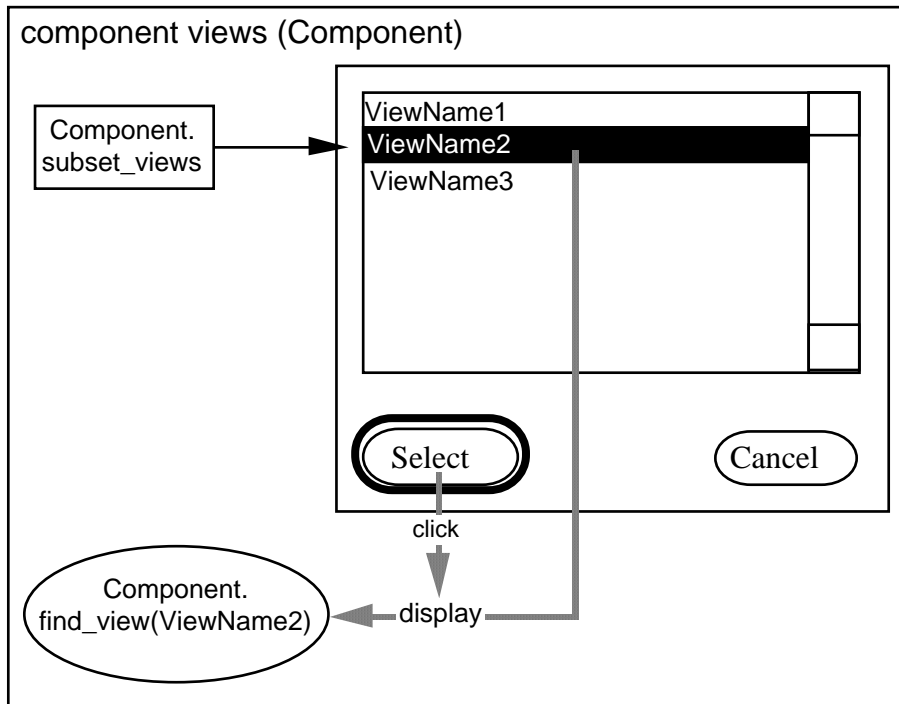
fig E.13.   View selection dialog for MViews.

Fig E.14. shows the features selection dialog for IspelM. Programmers can select a feature and perform various actions on the feature depending on the dialog button used.
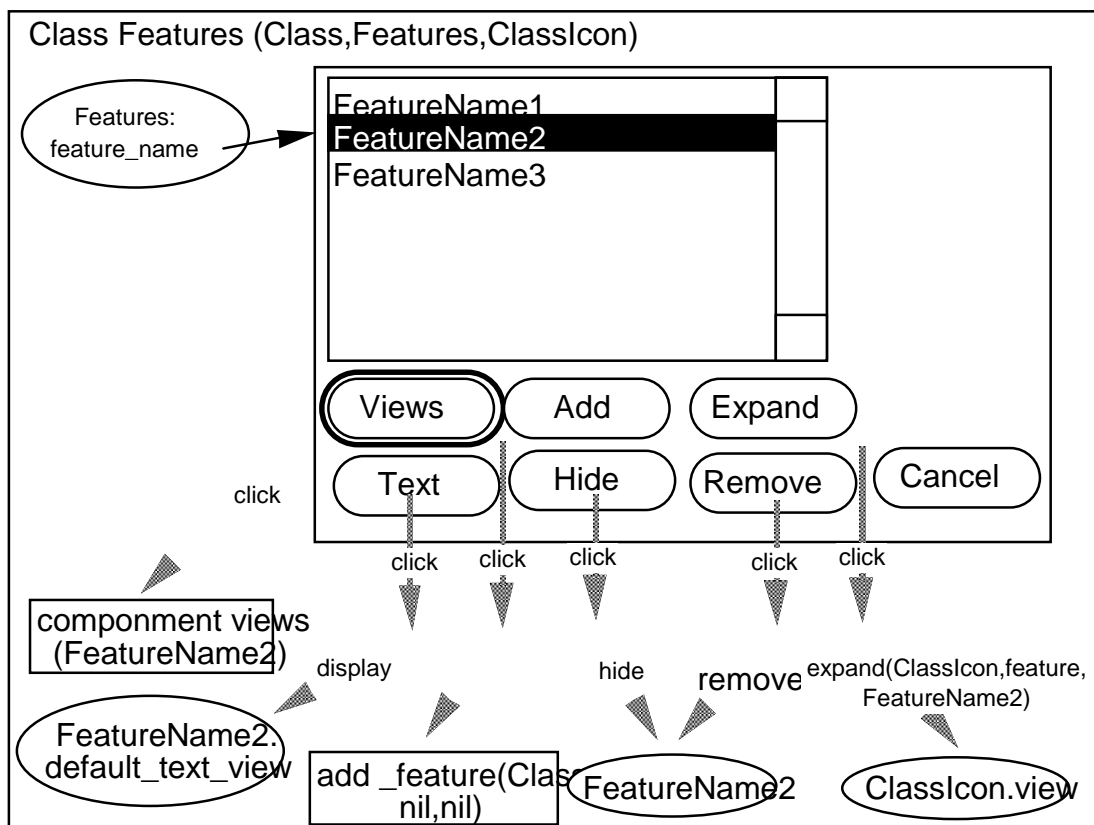

fig E.14.   The features selection dialog for IspelM.

Fig E.15. illustrates the add feature dialog for IspelM. This allows programmers to add features to a class (or change an existing feature) and specify/change various attributes of the feature.
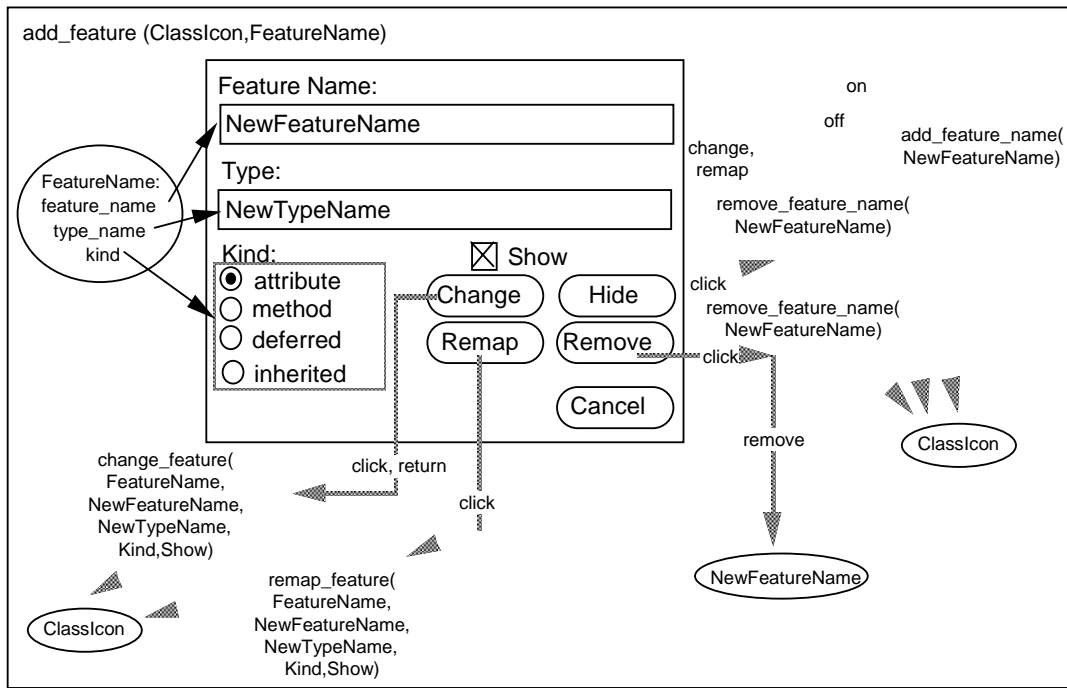

fig E.15.   The add feature dialog from IspelM.

Fig E.16. shows the generali class information dialog from IspelM. Like the features selection dialog, this allows programmers to select different class components and manipulate them or expand them into a view.
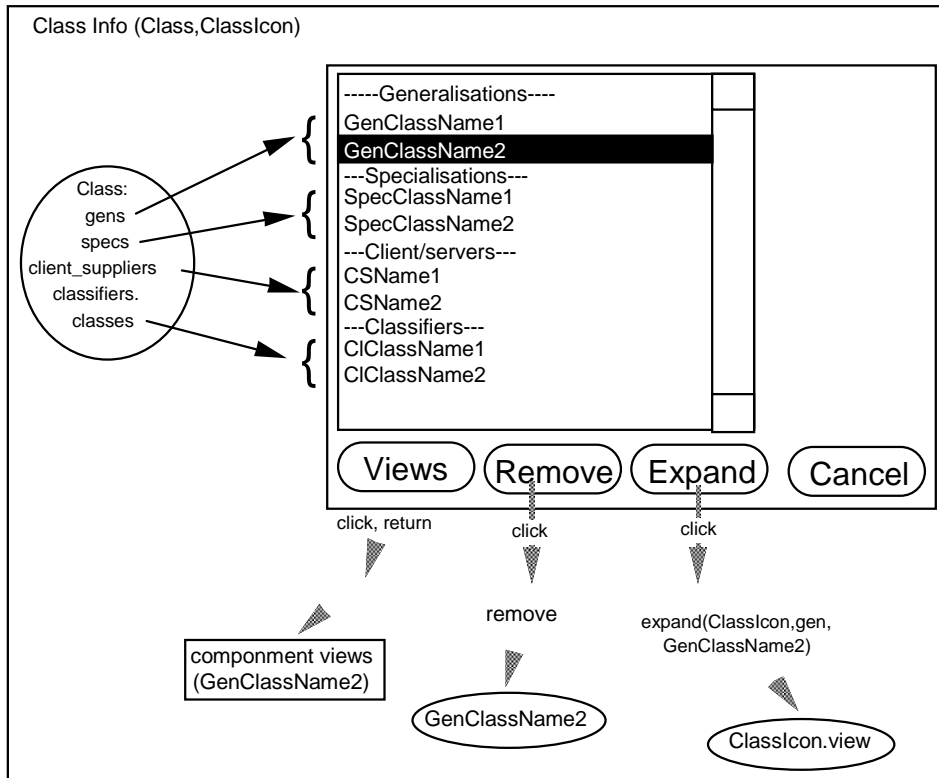
fig E.16.   The general class information dialog from IspelM.

Fig E.17. shows the client-supplier update details dialog. This is used to initially specify the attributes for a new client-supplier relationship and to update the details of an existing client-supplier relationship (or, using Remap, to find another base client-supplier relationship to represent).
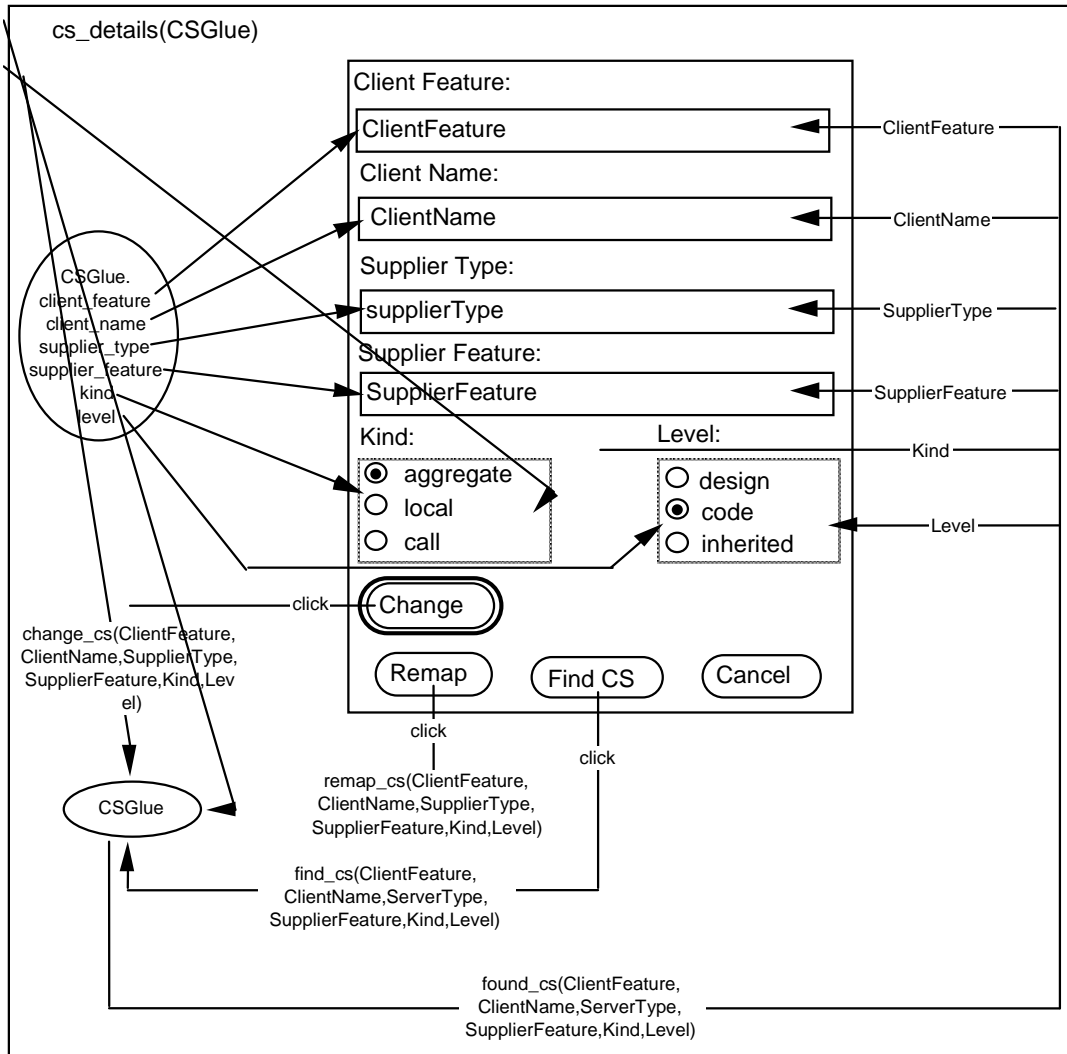
fig E.17.   The client-supplier update details dialog for IspelM.

Fig E.18. shows the classifier update details dialog. Like the client-supplier update details dialog, this allows programmers to initialise, update and remap classifier glue.
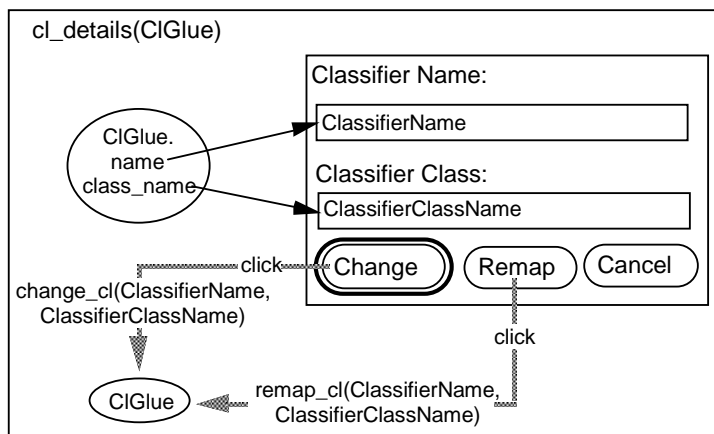

fig E.18.   The classifier update details dialog from IspelM.

Appendix F

A BNF Grammar for MVSL

```
Program ::=
    program Ide
        DeclList
        Command
    end Ide
    ;

DeclList ::=
    DeclList
    Decl
    | Decl
    | -- empty
    ;

Decl ::=
    BaseViewDecl
    | BaseElDecl
    | BaseRelDecl
    | SubsetViewDecl
    | SubsetElDecl
    | SubsetRelDecl
    | CompDecl
    | GlobalDecl
    ;

BaseViewDecl ::=
    base view Ide
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

BaseElDecl ::=
    base element Ide
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

BaseRelDecl ::=
    base relationship Ide
        ParentDecl
        ChildDecl
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

SubsetViewDecl ::=
    subset view Ide
        ComponentsDecl
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;
```

```
SubsetElDecl ::=
    subset element Ide
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

SubsetRelDecl ::=
    susbet relationship Ide
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

CompDecl ::=
    component Ide
        AttributesDecl
        RelationshipsDecl
        OperationsDecl
        UpdatesDecl
    end Ide
    ;

GlobalDecl ::=
    Ide ':' Type
    ;

ParentDecl ::=
    parent Ide
    ;

ChildDecl ::=
    child Ide
    ;

ComponentsDecl ::=
    components CompList
    ;

CompList ::=
    CompList
    Ide
    |
    Ide
    |
    -- empty
    ;

AttributeDecl ::=
    attributes AttributeList
    |
    -- empty
    ;

AttributeList ::=
    AttributeList
    Attribute
    | Attribute
```

```
    | -- empty
    ;

Attribute ::=
    Ide ':' Type
    ;

RelationshipDecl ::=
    relationships RelationshipList
    | -- empty
    ;

RelationshipList ::=
    RelationshipList
    Relationship
    | Relationship
    | -- empty
    ;

Relationship::=
    Ide ':' Type
    ;

OperationDecl ::=
    operations OperationList
    | -- empty
    ;

OperationList ::=
    OperationList
    Operation
    | Operation
    ;

Operation ::=
    Ide OpArgList OpType
    LocalDecl
    is
    Command
    end Ide
    ;

OpArgList :: =
    '(' OpArgs ')'
    | -- empty
    ;

OpArgs ::=
    OpArgs
    OpArg
    | OpArg
    ;

OpArg ::=
    in Ide ':' Type
    | out Ide ':' Type
    ;

LocalDecl ::=
    locals Locals
    | -- empty
    ;

Locals ::=
    Locals
    Local
    | Local
    ;

Local ::=
    Ide ':' Type
    ;

UpdateDecl ::=
    updates UpdateList
    | -- empty
    ;

UpdateList ::=
    UpdateList
    Update
```

```
    | Update
    ;

Update ::=
    Ide UpdArgList
    where Exp
    LocalDecl
    is
    Command
    end Ide
    ;

UpdArgList ::=
    '(' UpdArgs ')'
    | -- empty
    ;

UpdArgs ::=
    UpdArgs
    UpdArg
    | UpdArg
    ;

UpdArg ::=
    Ide ':' Type
    ;

Command ::=
    Exp := Exp
    | if Exp then Command else Command end
if
    | while Exp do Command end while
    | forall Exp on Exp do Command end
forall
    | add_element '(' Ide ',' Exp ')'
    | delete_component '(' Exp ')'
    | establish '(' Ide ',' Exp ',' Exp
',' Exp ')'
    | establish '(' Ide ',' Exp ',' Exp )
    | reestablish ( Exp ', Exp , Exp )
    | dissolve '(' Ide ',' Exp ',' Exp ')'
    | record '(' Exp , Ide ',' ExpList ')'
    | store '(' Exp , Ide ',' ExpList ')'
    | create_view '(' Ide ',' Exp ')'
    | add_view_component '(' Exp , Exp )
    | remove_view_component '(' Exp ','
Exp ')'
    | Exp
    | Exp '(' ExpList ')'
    ;

Exp ::=
    Integer
    | true
    | false
    | String
    | Ide
    | Exp '+' Exp
    | Exp '-' Exp
    | Exp '*' Exp
    | Exp '\' Exp
    | '(' Exp ')'
    | Exp and Exp
    | Exp or Exp
    | Exp '=' Exp
    | Exp '\==' Exp
    | Exp '.' Ide
    | Exp '(' ExpList ')'
    ;

ExpList ::=
    ExpList
    Exp
    | Exp
    ;

Type ::=
    integer
    | boolean
    | string
    | list Type
    | Ide
```

```
      Ide '.' Ide                                        |  'one-to-many' Type
      'one-to-one' Type                              ;
```

References

Ambler and Burnett, 89

Ambler, A., Burnett, M.M., Influence of Visual Technology on the Evolution of Language Environments, In *COMPUTER*, October, 1989, pp. 9-22.

Amor, 91

Amor, R., *ICAtect: Integrating Design Tools for Preliminary Architectural Design*, MSc Thesis, Department of Computer Science, Victoria University of Wellington, 1991.

Amor et al, 91

Amor, R., Groves, L. and Donn, M., Integrating Design Tools: An Object-Oriented Approach, Building Systems Automation, In *Integration '91*, June 2-8, Madison, Wisconsin, USA, 1991.

Amor et al, 92

Amor, R.A., Hosking, J.G., Groves, L.J., Donn, M.R., 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.

Apperley and Spence, 88

Apperley, M.D., Spence, R., *Lean Cuisine: A Low-Fat Notation for Menus*, Information Engineering Report #88/1, Department of Electrical Engineering, Imperial College of Science, Technology and Medicine, London, 1988.

Apple, 85

Apple Computer, *Inside Macintosh, Volume I*, Addison-Wesley, 1985.

Arefi et al, 90

Arefi F., Hughes C.E., and Workman D.A., Automatically Generating Visual Syntax-Directed Editors, In *Communications of the ACM*, Vol. 33, No. 3, 1990, pp. 349-360.

Attardi et al, 89

Attardi G., Bonini, C., Boscotrecase M. R., Flagella, T., and Gaspari, M., Metalevel Programming in CLOS, In *Proceedings of ECOOP '89 Conference*, Cambridge University Press, 1989, pp. 243-257.

Avrahami et al, 89

Avrahami, G., Brooks, K.P., Brown, M.H., A Two-View Approach to Constructing User Interfaces, In *ACM Computer Graphics*, Vol. 23, No. 3, July, 1989, pp. 137-146.

Backlund et al, 90

Backlund, B., Hagsand, O., Pehrson, B., Generation of Visual Language-oriented Design Environments, In *Journal of Visual Languages and Computing* 1, 4, 1990, pp.333-354.

Bailin, 89

Bailin, S.C., An Object-Oriented Requirements Specification Method, In *Communications of the ACM*, Vol. 32, No. 5., May 1989, pp. 608-623.

Borras et al 88

Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V., CENTAUR: the system, In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, November 1988.

Brooks, 87

Brooks, F.P., No silver bullet: essence and accidents in software engineering, In *COMPUTER*, Vol. 20, No. 1, January 1987, pp. 10-19.

Brown et al, 85

Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A., Souza, P., Program Visualisation: Graphical Support for Software Development, In *COMPUTER*, August 1985, pp. 27-35.

Brown, 88

Brown, M.H., Exploring algorithms using BALSA-II, In *COMPUTER*, Vol. 21, No. 5, May 1988, pp. 14-36.

Brown, 91

Brown, M.H., Zeus: A System for Algorithm Animation and Multi-View Editing, In *1991 IEEE Symposium on Visual Languages*, 1991, pp. 4-9.

Chen, 76

Chen, P.P., The Entity-Relationship Model - Toward a Unified View of Data, In *ACM Transactions on Database Systems*, Vol. 1, No. 1, March 1976, pp. 9-36.

Chikofsky and Rubenstein, 88

Chikofsky, E.T., Rubenstein, B.L., CASE: Reliability Engineering for Information Systems, In *IEEE Software*, March 1988, pp. 11-16.

Coad and Yourdon, 91

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

Consens and Mendelzon, 92

Consens, M., Mendelzon, A., Visulizing and querying software structures, In *14th International Conference on Software Engineering*, 1992, pp. 138-156.

Consens and Mendelzon, 93

Consens, M., Mendelzon, A., *Hy⁺: A Hygraph-based Query and Visualization System*, Technical Report, Computer Systems Research Institute, University of Toronto, 1993.

Cox et al, 89

Cox, P.T., Giles, F.R., Pietrzykowski, T., Prograph: a step towards liberating programming from textual conditioning, In *1989 IEEE Workshop on Visual Languages*, 1989, pp. 150-156.

Czejdo et al, 90

Czejdo, B., Elmasri, R., Rusinkiewicz, M., Embley, D.W., A Graphical Data Manipulation Language for an Extended Entity-Relationship Model, In *COMPUTER*, March 1990, pp. 26-36.

Dannenberg, 91

Dannenberg, R.B. A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors, In *Software-Practice and Experience*, Vol. 20, No. 2, February 1991, 109-132.

Dart et al, 87

Dart, S.A., Ellison, R.J., Feiler, P.H., Habermann, A.N., Software Development Environments, In *COMPUTER*, November, 1987, pp. 18-27.

Duke et al 91

Duke, R., King, P., Rose, G., Smith, G., *The Object-Z Specification Language Version 1*, Technical Report No. 91-1, Software Verification Research Centre, The University of Queensland, May 1991.

Fenwick, 93

Fenwick, S., *Cerno Implementation*, Preliminary Development Report, Department of Computer Science, 1993.

Fenwick and Hosking, 93

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

Fenwick, 94

Fenwick, S., *A Visual Debugger for Object-Oriented Programs*, MSc Thesis, Department of Computer Science, University of Auckland, 1994.

Fichman and Kemerer, 92

Fichman, R.G., and Kemerer, C.F., Object-oriented and Conventional Analysis and Design Methodologies, In *COMPUTER*, October 1992, pp. 22-39.

Finlay and Allison, 93

Finlay, A., and Allison, L., A Correction to the Denotational Semantics for the Prolog of Nicholson and Foo, In *ACM Transactions on Programming Languages and Systems*, Vol. 15 , No. 1, January 1993, pp. 206-208.

Fischer, 87

Fischer, G., Cognitive View of Reuse and Redesign, In *IEEE Software*, July 1987, pp. 60-72.

Garlan, 86

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

Garlan et al, 92

Garlan, D., Kaiser, G.E, Notkin, D., Using Tool Abstraction to Compose Systems, In *COMPUTER*, Vol. 25, No. 6, June 1992, pp. 30-38.

Glinert and Tanimoto, 85

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

Glinert, 89

Glinert, E.P., Towards software metrics for visual programming, In *International Journal of Man-Machine Studies*, Vol. 30, 1989, pp. 425-445.

Goldberg, 84

Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA., 1984.

Goldberg and Robson, 84

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

Golin and Reiss, 90

Golin, E.J., and Reiss, S.P., Specification of Visual Language Syntax, In *Journal of Visual Languages and Computing*, Vol. 1, No. 2., June 1990, pp. 141-157.

Gordon, 79

Gordon, M., *The Denotational Description of Programming Languages, An Introduction*, Springer-Verlag, 1979.

Grundy, 91

Grundy, J.C., *A Visual Programming Environment for Object-oriented Languages*, MSc Thesis, Department of Computer Science, University of Auckland.

Grundy et al, 91

Grundy, J.C., Hosking, J.G., Hamer, J., A Visual Programming Environment for Object-oriented Languages, in *Proceedings of TOOLS US '91*, Prentice-Hall, 1991, pp. 129-138.

Grundy and Hosking, 93

Grundy, J.C., and Hosking, J.G., The MViews framework for constructing multi-view editing environments, In *New Zeland Journal of Computing*, Vol. 4, No. 2, 1993, pp. 31-40.

Grundy and Hosking, 93

Grundy, J.C., and Hosking, J.G., The MViews framework for building visual programming environments, *Proceedings of the 1993 IEEE Symposium on Visual Languages*, 1993, pp. 220-224.

Haarslev and Möller, 90

Haarslev, V. and Möller, R., A Framework for Visualizing Object-Oriented Systems, in *Proceedings of ECOOP/OOPSLA '90*, October, 1990, pp. 237-244.

Hamer, 90

Hamer, J., *Expert Systems for Codes of Practice*, PhD Thesis, Department of Computer Science, University of Auckland, 1990.

Hamer et al, 92

Hamer, J., Hosking, J.G., Mugridge, W.B., *Static Subclass Constraints and Dynamic Class Membership Using Classifiers*, Departmental Report #62, Department of Computer Science, University of Auckland, 1992.

Henderson and Notkin, 87

Henderson, P.B., Notkin, D., Integrated Design and Programming Environments, In *COMPUTER*, November 1987, pp. 12-16.

Henderson-Sellers and Edwards, 90

Henderson-Sellers, B. and Edwards J.M., The Object-Oriented Systems Life Cycle, In *Communications of the ACM*, Vol. 33, No. 9, 1990, pp. 142-159.

Horowtiz and Teitelbaum, 86

    Horwitz, S. and Teitelbaum, T., Generating Editing Environments Based on Relations and Attributes, In *ACM TOPLAS*, Vol. 8, No. 4, 1986, pp. 577-608 .

Hosking et al, 90

    Hosking, J.G., Hamer, J., Mugridge W.B., Integrating functional and object-oriented programming, In *Proceedings of TOOLS Pacific '90 Conference*, Sydney, August 1990, Prentice-Hall, pp. 345-355.

Hudson, 91

    Hudson, S.E., Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update, In *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 3, July 1991, pp. 315-341.

Ingalls et al, 88

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

Interactive, 89

    Interactive Software Engineering, *Eiffel: The Environment*, Technical Report TR-EI-5/UM (Version 2.2), Interactive Software Engineering Inc., August 1989.

Jones, 92

    Jones, M.P., *Gofer: An Introduction to Gofer*, Technical Report, Programming Research Group, Oxford University Computing Laboratory, February 1992.

Kaiser, 85

    Kaiser, G.E., *Semantics for Structure Editing Environments*, PhD Thesis, Department of Computer Science, Carnegie-Mellon University, CMU-CS-85-131, 1985.

Kaiser and Garlan, 87

    Kaiser, G.E, Garlan, D., Melding Software Systems from Reusable Blocks, In *IEEE Software*, Vol. 4, No. 4, July 1987, pp. 17-24.

Kaiser et al, 87

    Kaiser, G.E., Kaplan, S.M., Micallef, J., Multiuser, Distributed Language-Based Environments, In *IEEE Software*, November 1987, pp. 58-67.

Keene, 89

    Keene, S.E., *Object-oriented Programming in Common LISP: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

Kleyn and Gingrich, 88

Kleyn, M.F., Gingrich, P.C., GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views, In *Proceedings of OOPSLA '88*, ACM, pp. 191-205.

Kleyn and Browne, 93

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

LaLonde and Pugh, 93

LaLonde W., Pugh, J., Instance-based programming with PARTS, In *Journal of Object-Oriented Programming*, March-April 1993, pp. 75-81.

Langerak, 90

Langerak, R., View Updates in Relational Databases with an Independent Scheme, In *ACM Transactions on Database Systems*, Vol. 15, No. 1, 1990, pp. 40-66.

Larson, 86

Larson, J.A., A Visual Approach to Browsing in a Database Environment, In *COMPUTER*, Vol. 19, No. 6, June 1986, pp. 62-71.

Leidig and Mühlhäuser, 91

Leidig, T., and Mühlhäuser, M., *An Object-Oriented Graphical Editor for Distributed Application Development*, Technical Report, Department of Informatics, University of Kaiserslautern, 1991.

Lieberman, 86

Lieberman, H., Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems, In *Proceedings of OOPSLA '86*, ACM, pp. 214-223.

Linton et al, 88

Linton, M.A., Vlissides, J.M., Calder, P.R., Composing User Interfaces with Interviews, In *COMPUTER*, February 1989, pp. 8-22.

LPA, 92

Logic Programming Associates, *LPA Prolog Version 4.5 Reference Manual*, Logic Programming Associates, 1992.

Magnusson et al, 90

Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minör, S., Oscarsson, D., Taube, M., An Overview of the Mjølner/ORM

Environment: Incremental Language and Software Development, In *Proceedings of TOOLS '90*, Prentice-Hall.

Mannucci et al, 89

Mannucci, S., Mojana, B., Navazo, M.C., Romano, V., Terzi, M.C., Torrigiani, P., Graspin: A Structured Development Environment for Analysis and Design, In *IEEE Software*, November 1989, pp. 35-43.

Malenfant et al, 89

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

McIntyre and Glinert, 92

McIntyre, D., Glinert, E., Visual Tools for Generating Iconic Programming Environments, In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, IEEE Press, 1992, pp. 162-168.

Meyer, 87

Meyer, B., Reusability: The Case for Object-Oriented Design, In *IEEE Software*, March 1987, pp. 50-64.

Meyer, 88

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

Meyer, 92

Meyer, B., Applying "Design by Contract", In *COMPUTER*, October 1992, pp. 40-52.

Meyers, 91

Meyers, S., Difficulties in Integrating Multiview Editing Environments, In *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 49-57.

Minör, 90

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

Monarchi and Puhr, 92

Monarchi, D.E., Puhr, G.I., A Research Typology for Object-Oriented Analysis and Design, In *Communications of the ACM*, Vol. 35, No. 9, 1992, pp. 35-47.

Moon, 86

Moon, D.A., Object-oriented Programming with Flavors, In *Proceedings of OOPSLA '86*, ACM, pp. 1-8.

Myers et al, 88

Myers, B.A., Chandhok, R., Sereen, A., Automatic Data Visualisation for Novice Pascal Programmers, In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, October 1988, IEEE, pp. 192-198.

Myers, 89

Myers, B.A., User interface tools: introduction and survey, In *IEEE Software*, Vol. 6, No. 1, January 1989, pp. 15-23.

Myers, 90

Myers, B.A., Taxonomies of visual programming and program visualization, In *Journal of Visual Languages and Computing*, Vol. 1., No. 1, March 1990, pp. 97-123.

Myers, 90

Myers, B.A., Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, In *COMPUTER*, Vol. 23, No. 11, 1990, pp. 71-85.

Nascimento and Dollimore, 93

Nascimento, C., and Dollimore, J., A model for co-operative object-oriented programming, In *Software Engineering Journal*, Vol. 8, No. 1, January 1993, pp. 41-48.

Newbury, 88

Newbury, F.J., An Interface Description Language for Graph Editors, In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, IEEE, pp. 144-149.

Noble and Groves, 92

Noble, R.J., and Groves, L., Tarraingim: A Program Animation Environment, In *New Zealand Journal of Computing*, Vol. 4, No. 1, 1992, pp. 29-41.

Notkin 85

Notkin, D., The GANDALF Project, In *The Journal of Systems and Software*, Vol. 5, No. 2., May 1985.

O'Brien et al, 87

O'Brien, P.D., Halbert, D.C., Kilian, M.F., The Trellis Programming Environment, In *Proceedings of OOPSLA '87*, ACM, 1987, pp. 91-102.

Olsen and Dempsey, 83

Olsen, D.R, Dempsey, E.P., Syngraph: A Graphical User Interface Generator, In *Proceedings of ACM SIGGRAPH '83 Conference*, July 1983, pp. 43-50.

Paulisch and Tichy, 90

Paulisch, F.N., Tichy, W.F., EDGE: An Extensible Graph Editor, In *Software - Practice and Experience*, Vol. 20, No. S1, June 1990, pp. S1/63-S1/88.

Pountain, 90

Pountain, D., Adding Objects to Prolog, In *Byte*, August 1990, pp. 64IS-15-64IS20.

Quintus, 91a

Quintus Corporation, *Quintus Prolog Reference Manual*, Quintus Corporation, Palo Alto, CA, 1991.

Quintus, 91b

Quintus Corporation, *ProTALK Reference Guide*, Quintus Corporation, Palo Alto, CA, 1991.

Raeder, 85

Raeder, G., A Survey of Current Graphical Programming Techniques, In *COMPUTER*, August 1985, pp. 11-25.

Ratcliff et al, 92

Ratcliff, M., Wang, C., Gautier, R.J., Whittle B.R., Dora - a structure oriented environment generator, In *Software Engineering Journal*, Vol. 7, No. 3, 1992, pp. 184-190.

Reiss, 85

Reiss, S.P., PECAN: Program Development Systems that Support Multiple Views, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 3, March 1985, pp. 276-285.

Reiss, 86

Reiss, S.P., GARDEN Tools: Support for Graphical Programming, In *Proceedings of Advanced Programming Environments*, Trondheim, Norway, June 1986, Lecture Notes in Computer Science #244, Springer-Verlag, pp. 59-72.

Reiss, 87

Reiss, S.P., Working in the GARDEN Environment for Conceptual Programming, In *IEEE Software*, November 1987, pp. 16-26.

Reiss, 90a

Reiss, S.P., Connecting Tools Using Message Passing in the Field Environment, In *IEEE Software*, July 1990, pp.57-66.

Reiss, 90b

Reiss, S.P., Interacting with the FIELD Environment, In *Software - Practice and Experience*, Vol. 20, No. S1, June 1990, pp. S1/89-S1/115.

Reps and Teitelbaum, 84

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

Reps and Teitelbaum, 87

Reps, T., Teitelbaum, T., Language Processing in Program Editors, In *COMPUTER*, November, 1987, pp. 29-40.

Rosenblatt et al, 89

Rosenblatt, W.R., Wileden, J.C., Wolf, A.L., OROS: Toward a Type Model for Software Development Environments, In *Proceedings OOPSLA '89*, ACM, 1989, pp. 297-304.

Rumbaugh et al 91

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W, *Object-oriented Modeling and Design*, Prentice-Hall, 1991.

Sajeev and Hurst, 92

Sajeev, A.S.M., Hurst, A.J., Programming Persistence in χ, In *COMPUTER*, Vol. 25, No. 9, September 1992, pp. 57-66.

Santucci and Sottile, 93

Santucci, G., Sottile, P.A., Query by Diagram: a Visual Environment for Querying Databases, In *Software-Practice and Experience*, Vol. 23, No. 3, pp. 317-340.

Shlaer and Mellor, 88

Shlaer, S., and Mellor, S.J., *Object-Oriented Analysis: Modeling the World in Data*, Yourdon Press, Englewood Cliffs, N.J., 1988.

Stroustrup, 86

Stroustrup, B., *The C++ programming language*, Addison-Wesley, 1986.

Stasko, 89

Stasko, J.T., *TANGO: A Framework and System for Algorithm Animation*, PhD Dissertation, Department of Computer Science, Brown University, Providence, Rhode Island, TR CS-89-30, 1989.

Staudt et al, 88

Staudt, B., Krueger, C., Garlan, D., *TransformGen: Automating the Maintenance of Structure-Oriented Environments*, Technical Report CMU-CS-88-186, Department of Computer Science, Carnegie-Mellon University, 1988.

StructSoft, 92

StructSoft Inc, *TurboCASE*, StructSoft Inc, 5416 156th Ave. S.E. Bellevue, WA 98006, 1992.

Symantec, 90

Symantec Corporation, *THINK C Reference Manual*, Symantec Corporation, 1990.

Tammassia et al, 88

Tammasia, R., Battista, G. D., Batini, C., Automatic Graph Drawing and Readability of Diagrams, In *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 18, No. 1, January/February 1988, pp. 61-79.

Tennent, 76

Tennent, R.D., The Denotational Semantics of Programming Languages, In *Communications of the ACM*, Vol. 19, No. 8, August 1976, pp. 437-453.

Teorey et al, 86

Teorey, T. J., Yang, D., and Fry, J.P., A logical design methodology for relational databases using the extended Entity-Relationship model, In *ACM Computing Surveys*, Vol. 18, No. 2, June 1986, pp. 197-222.

Teorey et al, 89

Teorey, T.J., Guangping, W., Bolton, D.L., Koenig, J.A., ER Model Clustering as an Aid for User Communication and Documentation in Database Design, In *Communications of the ACM*, Vol. 32, No. 8, August 1989, pp. 975-987.

Ungar et al, 92

Ungar, D., Smith, R.B., Chambers, C., Hölzle, U., Object, Message, and Performance: How They Coexist in Self, In *COMPUTER*, October 1992, pp. 53-64.

Vlissides, 90

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

Wang et al, 92

Wang, C., Leung, C-C., Ratcliffe, M., and Long, F., *Multiple Views of Software Development*, Technical Report, Computer Science Department, University College of Wales, Aberystwyth, 1992.

Wasserman, 89

Wasserman, A.I., Tool integration in software engineering, In *Lecture Notes in Computer Science*, Vol. 467, Springer-Verlag, 1989.

Wasserman and Pircher, 87

Wasserman, A.I., Pircher, P.A., A Graphical, Extensible, Integrated Environment for Software Development, In *SIGPLAN Notices*, Vol. 22, No. 1, January, 1987, pp. 131-142.

Wasserman et al, 90

Wasserman, A.I., Pircher, P.A., Muller, R.J., The Object-oriented Structured Design Notation for Software Design Representation, In *COMPUTER*, March 1990, pp. 50-63.

Welsh et al, 91

Welsh, J., Broom, B., Kiong, D., A Design Rationale for a Language-based Editor, In *Software - Practice and Experience*, Vol. 21, No. 9, September 1991, pp. 923-948.

Whittle et al 92

Whittle, B.R., Gautier, R.J., Ratcliffe, M., *Trends in Structure Oriented Environments*, Technical Report UCW-SEG-601-92, University Colledge of Wales, Abersystwyth, August 1992.

Wilk, 91

Wilk, M.R., Change Propagation in Object Dependency Graphs, In *Proceedings of TOOLS US '91*, Prentice-Hall 1991, pp. 233-247.

Wilson, 90

Wilson, D.A., Class Diagrams: A Tool for Design, Documentation and Teaching, In *Journal of Object-Oriented Programming*, January/February 1990, pp. 38-44.

Winblad et al, 90

Winblad, A.L., Edwards, S.D., King, D.R., *Object-Oriented Software*, Addison-Wesley, 1990.

Wirfs-Brock and Wilkerson, 89

Wirfs-Brock, R., Wilkerson, B., Object-Oriented Design: A Responsibility-Driven Approach, In *Proceedings of OOPSLA '89*, ACM, 1989, pp. 71-75.

Yourdon, 89

Yourdon, E., *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, New Jersey, 1989.