SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(7), 773–798 (JUNE 1998)

Constructing an Integrated Visual Programming Environment

chung-hua hu and feng-jian wang
National Chiao-Tung University, Department of Computer Science and Information Engineering, Taiwan, 30050, R.O.C.
(email: {chhu,fjwang}@csie.nctu.edu.tw)

SUMMARY

This paper presents an object-oriented architecture, called the Model-View-Shape (MVS) architecture, for constructing an Integrated Visual Programming Environment (IVPE), whose constituent tools deal with (fine-grained) language semantics, as well as a mass of graphics-drawing activities. This architecture enforces a layered and loosely-coupled structure, so that the user-interface part of components may be more independent, maintainable, and reusable than those proposed in the original model-view-controller architecture. An MVS class hierarchy, systematically constructed using C++, can be reused and extended with new semantics to rapidly develop new tools for an existing IVPE, or even an IVPE supporting more than one language. The present editors developed can be used to construct programs by specifying the associated flow information in explicit (visual) or implicit (textual) ways, while the (incremental) flow analysers can help analyse incomplete program fragments to locate and inform the user of possible errors or anomalies during programming. © **1998 John Wiley & Sons, Ltd.**

key words: visual programming; integrated programming environment; object-oriented technique; reusability; C++

INTRODUCTION

Visual programming, referring to any system that allows the user to specify programs in a multi-dimensional style,[1] is intended to ease the programming process through simplicity, concreteness, explicitness, and responsivenes.[2] Many studies concerning visual programming can be found in the literature.[3–5] On the one hand, *visual programming languages*[6,7] are designed to help construct programs using visual language constructs. On the other hand, *Visual Programming Environments* (VPE)[8,9] which consist of a wide variety of program-development tools, have been constructed. These tools usually employ graphical techniques, such as *direct manipulation*,[10] to manipulate pictorial elements and to display the structure of programs. In addition, a programming environment is called *integrated*[11–13] when the constituent tools in the environment share consistent programming information, and interact with the user through a uniform user interface.

Most studies of VPEs have focused on the functional descriptions of these environments. A few studied how to construct or generate VPEs in a systematic manner.[14,15] In this paper, a *Model-View-Shape* (MVS) architecture, adapted from

CCC 0038–0644/98/070773–26$17.50
© 1998 John Wiley & Sons, Ltd.
Received 2 July 1997
Revised and accepted 19 February 1998

the *Model-View-Controller* (MVC) architecture,[16] is used to construct an Integrated VPE (IVPE) that, in particular, needs to deal with program semantics as well as a mass of graphics-drawing activities. In our approach, the grammar of a target language is modeled as a suite of software components embodying fine-grained language semantics and presentation information. As Figure 1 shows, an IVPE can be divided into three main modules: *application*, *user-interface*, and *graphics-handling*. The application module, a set of model objects, is responsible for handling program semantics as well as language-dependent analysis. The user-interface module, a set of view objects, is responsible for managing (high-level) program presentation in visual or textual ways. The graphics-handling module, a set of shape objects, is responsible for drawing (low-level) graphical primitives and handling/interpreting user-input events. These objects are cooperative; they communicate with each other via message-passing to complete program analysis and graphics display for each editing action. The MVS architecture enforces a layered and loosely-coupled structure, so that components in the user-interface part may be more independent, maintainable, and reusable than that proposed in the original MVC architecture.

During the construction of our IVPE, a C++ class hierarchy based on the MVS architecture was first systematically constructed. The MVS class hierarchy was then reused and extended to construct new tools for the IVPE. For those front-end tools which need to provide a variety of graphics-drawing facilities, shape classes are reusable because they are application-independent; while model and view classes may be augmented with new functionalities. Construction of back-end tools, such as semantic and data-flow analysers, is limited to the refinement and addition of *semantic attributes* and *evaluation methods* to the model classes in the MVS class hierarchy. Moreover, the MVS class hierarchy constructed may be considered generic for different programming languages. To support more than one language in an IVPE, designers can create model (and view) classes for new language constructs, and add them to the MVS hierarchy by locating appropriate base classes. In this way, generic attributes and methods are reused, while language-specific features are implemented in the corresponding new classes.

Our current IVPE provides well-designed editing, display, and analysis facilities

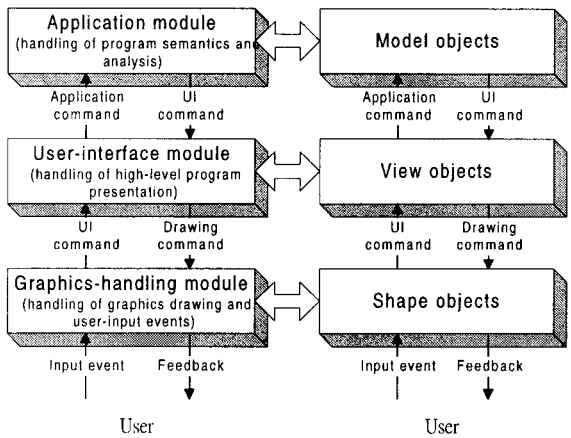

Figure 1. The model-view-shape architecture

that ease program development. For example, a flow-based and syntax-directed[17,18] editing model associated with *zooming* and *folding* facilities help construct programs by depicting control-flow graphs. The (incremental) flow analysers can work on incomplete program fragments to locate and inform the user of possible errors or anomalies during programming.

REQUIREMENTS

As software and visual technology evolves, dozens of requirements related to programming environments have been successively presented in the literature. Here we summarize six features that seem essential to practical IVPEs. The first four features refer to methodologies that enable users to construct and display programs in an IVPE, while the rest are concerned with extension and evolution aspects of the IVPE itself.

(a) *Support for visual and textual program construction*.[19] Visual programming is certainly one of the core functionalities for all IVPEs. However, a number of studies[20,21] show that pure visual representations may be less understandable than textual representations in some cases. To compensate for a potential shortage of visual programming, an IVPE should support both visual and textual tools that can be invoked on demand to display whatever representations the user wants to work on.

(b) *Support for incremental program development*.[18] Incremental program development, in general, has two advantages: (1) the analysis and execution of incomplete programs are possible; and (2) programming errors can be detected earlier than by employing compilation technology alone in conventional program development.

(c) *Support for multiple and consistent views of programs*.[19,22,23] The ability to see multiple views helps the user understand various parts of a program concurrently. To maintain consistency among multiple views when one view is modified, all other views that share the modified program text are informed of changes so they can update the outputs correspondingly.

(d) *Effective management of large-program displays*.[3] The graphical layout of a program displayed in an IVPE usually consumes more screen space. This problem, called the *scalability problem*,[24] apparently impedes users' understanding processes when handling large programs. An IVPE may provide more than one simple way for the user to specify and control the graphical layout. When the user works on some part(s) of a program, the IVPE allows one to retrieve the part with the desired graphical layout, while unneeded parts are hidden from view automatically.

(e) *Support for tool integration and communication*.[11,25] Within an IVPE, the constituent tools cooperate and communicate with each other through a common interface and internal program representations.[26] In addition, there should be no need for the IVPE to be reconstructed or perform a large-scale modification when integrating a new tool.

(f) *Support for extension and customization of an IVPE*.[11,12] An IVPE is more usable if it provides methods or guidance for extending and customizing itself in order to meet different users' demands. For example, the IVPE should be easily extensible to support a multilingual programming environment.

SYSTEM ARCHITECTURE

Our IVPE, whose system architecture is shown in Figure 2, is proposed to meet the above design requirements. Tools in an IVPE can be categorized into three toolsets based on the functions they perform: *programming*, *maintenance*, and *analysis* toolsets. The first two toolsets, which interact with users during various phases of software development, are equipped with graphical user interfaces that display various kinds of program information. For example, they are capable of receiving user-input events, interpreting and handling these events, and responding to users with some feedback, such as reporting error messages. The analysis toolset, which consists of tools for handling and analysing program text during programming, can be viewed as back-end tools. These tools do not interact with users directly, but can be activated automatically once changes occur in internal program representations, such as program trees and symbol tables. These tools can be further classified into two categories: *incremental* and *non-incremental* tools. Sample incremental tools include the semantic analyser, the data-flow analyser, and even the code generator.

To ensure concurrency control of internal program representations in a consistent manner, all tools in the IVPE are prevented from operating on the shared represen-

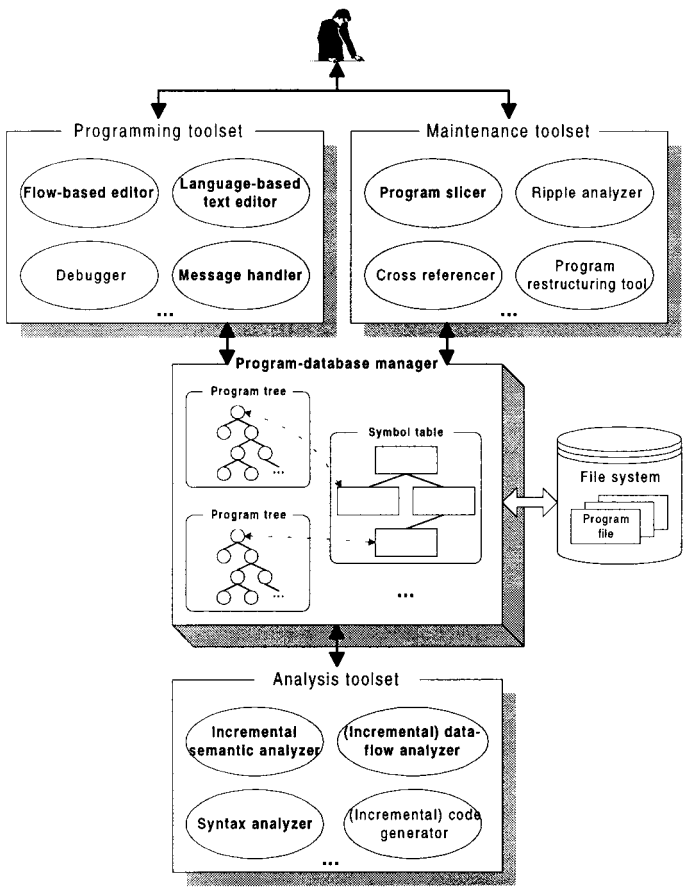

Figure 2. The system architecture

tations directly. The only way they can access the shared representations is through the program-database manager, which is responsible for coordinating tool communication, and maintaining data consistency by interacting with the underlying file system. So far we have constructed a number of tools, whose names are in boldface in Figure 2, in our IVPE.

DESIGN AND IMPLEMENTATION ASPECTS

The model-view-shape class hierarchy

A class hierarchy based on the MVS architecture was designed for constructing the kernel of our IVPE. As Figure 3 shows, the **`Model`**, **`View`**, and **`Shape`** class hierarchies correspond to model, view, and shape classes, respectively. The functionality and design issues of the MVS architecture are discussed below.

The shape classes

The IVPE allows the user to depict program flow information pictorially, thus it requires a collection of graphical components, such as nodes and links, to manipulate

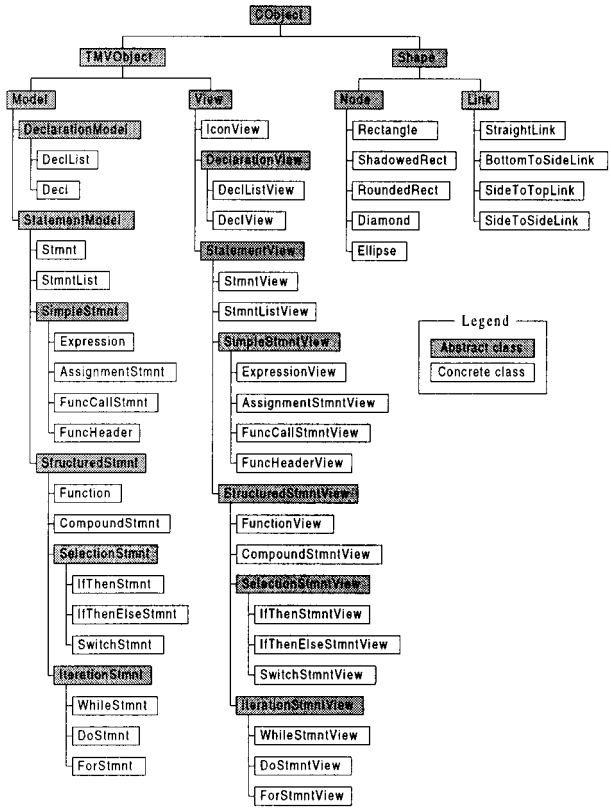

Figure 3. An MVS class hierarchy

visual presentation. A node denotes the notion of an entity, whereas a link, associating one node with another, describes the relationship between these two nodes. From the object-oriented viewpoint, all nodes and links can be treated as shape objects, so that each kind of graphical component can be defined as belonging to a shape class. Attributes defined in shape classes are used to store graphical layouts and related information, such as the dimensions and coordinates of graphical components. Methods defined in shape classes can be categorized into the following two sets:

(a) *Graphics-handling methods*. Examples include drawing graphical layouts at specific locations.
(b) *Event-handling methods*. Examples include detecting and interpreting user-input events.

The main design issues concerning shape classes involve: (1) How many kinds of graphical components need be constructed as shape classes? (2) How can new graphical components be added easily to the existing component library, i.e. the **`Shape`** class hierarchy? (3) How can graphical components be made general enough so they can be reused in constructing other flow-based applications?

In our construction approach, the **`Shape`** class hierarchy consists of two subclass hierarchies; one for node classes and the other for link classes. Common attributes and generic graphics-handling methods for these nodes and links are defined in classes **`Node`** and **`Link`**, respectively. On the basis of the **`Shape`** class hierarchy, new graphical components can be constructed as customized subclasses that inherit attributes and methods defined in the base class(es). The graphical components specified in the **`Shape`** class hierarchy are also application-independent, i.e. they are thought to be reusable elements.

The model and view classes

Model and view objects are responsible for managing the application's data structures and presentation, thus they may be viewed as *application-dependent* objects. For different flow-based applications, such as control-flow or state-transition diagram editors, different model and view classes may be identified and constructed. Application semantics, usually specified via attributes associated with handling methods, are then stored in the model and view classes. The main design issues concerning model and view classes involve: (1) How many kinds of model and view classes need be identified and constructed? (2) What methodology (or guidance) should be employed to classify these model and view classes into class hierarchies in a systematic and effective manner?

Our construction methodology for model classes is described below:

1. A model class is constructed for each kind of language construct defined in a target language. This procedure is referred to as *class assignment*.[27] For example, our IVPE enables users to construct and maintain programs in a C language subset, the associated context-free grammar (CFG) of which is listed in Table I. Each non-terminal symbol appearing in the CFG is represented as a specific class. Attributes defined in model classes are generally classified into two sets: one for the maintenance of internal program representations, such as program trees, and the other for storage of language-dependent information, such as source code, comments, and static semantics. Methods defined in model classes

are generally used to perform syntactic and semantic analyses, as well as language-dependent functions, such as setting the source code and comments.

2. The model classes constructed are then classified into a hierarchy according to two criteria: *OR* operators ('|')[11,27] in the grammar rules, and the language-construct functionality. The OR-operator criterion specifies that if a grammar rule looks like $\mathbf{X_0 ::= X_1} \mid \ldots \mid \mathbf{X_k}$ ($\mathbf{X_0}$, $\mathbf{X_1}$, and $\mathbf{X_k}$ denotes non-terminal symbols), then $\mathbf{X_0}$ can be constructed as a base class of $\mathbf{X_1}$, …, and $\mathbf{X_k}$. For example, it is appropriate to classify the **SelectionStmnt** class as a base class of **IfThenStmnt** and **IfThenElseStmnt** classes. On the basis of this criterion, the model class hierarchy can be *automatically* generated from the grammar rules of a target language. This scheme, however, has the following deficiencies. First, the class hierarchy constructed may be unnatural. For example, according to the CFG in Table I, the **AssignmentStmnt** class is a kind of the **Stmnt** class, which is a kind of the **StmntList** class. This may make **AssignmentStmnt** to be a kind of **StmntList** also. Second, some language constructs cannot be classified because they lack suitable base classes; the **Expression** class shows such an example. Third, this classification scheme may result in model classes of lower reusability because different languages may be defined via different grammar rules. Each time the IVPE is constructed to support a new language, a new model class hierarchy may need to be constructed.

To overcome the above deficiencies, an additional criterion, the functionality criterion, is introduced to perform a more complete and reasonable classification of language constructs manually. During the classification process, language constructs with common functions usually have common or very 'similar' attributes and methods defined in the corresponding model classes. For example, **AssignmentStmnt** and **Expression** classes both consist of identifiers and operators, so that they may be classified as being derived from the **SimpleStmnt** class, used for abstracting their common attributes and methods. In summary, the above two criteria provide guidelines for constructing a potentially reusable and extensible model class hierarchy in a *semi-automatic* manner.

The following describes our construction methodology for view classes.

1. A view class is constructed for each model class, and usually has *aggregation relationships* with one or more than one shape class. That is, a view object consists of a single or a set of shape objects for graphical presentation. Attributes defined in view classes are used to store high-level presentation

Table I. The CFG of a C language subset

```
⟨function⟩ ::= ⟨func-header⟩ ⟨decl-list⟩ ⟨compound-stmnt⟩
⟨decl-list⟩ ::= ⟨decl⟩ | ⟨decl⟩ ⟨decl-list⟩
⟨compound-stmnt⟩ ::= { ⟨stmnt-list⟩ }
⟨stmnt-list⟩ ::= ⟨stmnt⟩ | ⟨stmnt⟩ ⟨stmnt-list⟩
⟨stmnt⟩ ::= ⟨assignment-stmnt⟩ | ⟨selection-stmnt⟩ | ⟨iteration-stmnt⟩ | ⟨compound-stmnt⟩
⟨selection-stmnt⟩ ::= ⟨if-then-stmnt⟩ | ⟨if-then-else-stmnt⟩ | ⟨switch-stmnt⟩
⟨iteration-stmnt⟩ ::= ⟨while-stmnt⟩ | ⟨do-stmnt⟩ | ⟨for-stmnt⟩
⟨if-then-else-stmnt⟩ ::= If (⟨expression⟩) ⟨stmnt⟩ else ⟨stmnt⟩
```

information, such as view dimensions, and methods (defined in a view class) can be classified into the following two sets:

(a) *View-management methods*. Examples include calculating and retrieving view dimensions.

(b) *View-presentation methods*. Examples include presenting view layouts.

2. The view classes constructed are then classified into a hierarchy based on the structure of the **`Model`** class hierarchy.

By following the construction methodologies mentioned above, model and view classes representing different kinds of language constructs can be systematically classified into the corresponding hierarchies, as shown in Figure 3. Moreover, the MVS class hierarchy constructed may be considered generic for different programming languages. Designers can create model (and view) classes for new target language constructs, and add them to the existing hierarchy by locating appropriate base classes. In this way, new model (and view) classes automatically inherit all the properties, including those generic attributes and methods, from their base classes.

Graphical representations of language constructs

A set of graphical templates, as shown in Figure 4, have been designed to represent well-known language constructs using the syntax employed in the C language subset. These graphical templates are composed of existing graphical components (i.e. shape objects). For example, an if-then-else statement template consists of four nodes and four links. These graphical templates, which are the building blocks for program construction, can be treated as graphical extensions of conventional text-based programming language constructs. The whole programming

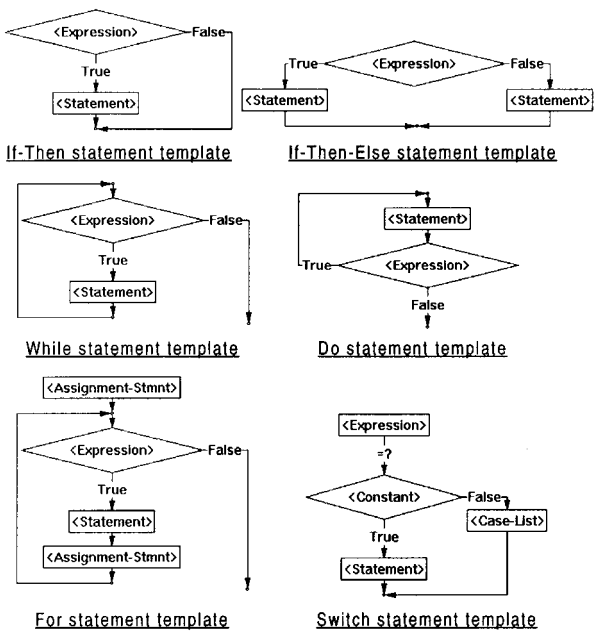

Figure 4. Sample graphical templates for language constructs

activities, similar to *algorithmic programming*,[3] can be divided into three phases: (1) selecting a graphical template, (2) placing it in the proper position on the screen, and (3) connecting the template to other templates via links that depict the desired control-flow information. Our editors handle tasks of phases 2 and 3 automatically in order to maintain readability.

Construction and manipulation of internal program representations

As users construct programs, our IVPE employs two data structures, *symbol tables* and *program trees*, to store these programs internally. To enable different tools to have consistent views of shared data structures, the program-database manager in our IVPE performs three functions. First, it manages internal program representations, that is, it is responsible for constructing and maintaining symbol tables and program trees. Second, it can be viewed as a *message server* for tool communication and coordination. Through the *service routines* provided by the program-database manager, tools in the IVPE are able to communicate with each other and access internal program representations concurrently and consistently. Third, it serves as an agent for storing and retrieving programs by interacting with the underlying file system. The following briefly describes the first two functions, while the associated implementation details can be found in Hu and Wang.[28]

The structure of a program tree is similar to that of an *abstract syntax tree*,[29] that is, each node in the program tree represents a specific kind of language construct, such as a statement or an expression. As the user constructs a program by inserting language constructs, appropriate new nodes are created and added to the program tree. In our construction approach, each tree node is represented by a *composite object*, called the template-based model-view (TMV for short) object. The term ‘composite’ means that a TMV object, which is just a conceptual notion, encapsulates one model object and one (or more than one) view object as its components.

Interactions among multiple TMV objects

Figure 5 shows a sample program tree representing an if-then-else statement, and illustrates the association relationships among model, view, and shape objects. The related attributes and the associated model and view classes for constructing such a program tree are listed in Table II. Each model object maintains two attributes called **`ParentModel`** and **`ChildModelList`** to reference the parent and child model objects, respectively. Each view object also contains similar attributes to maintain parent–child relationships with other view objects. Moreover, a model object usually has a default view object associated with it. When a model object is created, the default view object is created next. The creation sequence of the model, view, and shape objects for an if-then-else statement is shown in Figure 6.

As mentioned above, each program-tree node is represented by a TMV object, so that a program under construction is modeled by multiple TMV objects in hierarchical tree structures. These TMV objects are cooperative; they communicate with each other via message-passing between parent and child TMV objects to complete display tasks and incremental analysis for all editing actions. For example, when an if-then-else statement is about to be displayed, the **`IfThenElseStmntView`** object sends a message called **`OnDraw()`** to inform all its child view objects to display ‘themselves’.

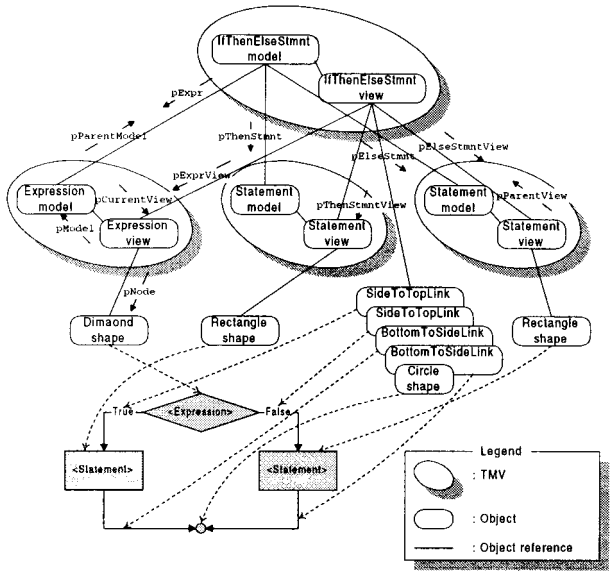

Figure 5. Relationships among model, view, and shape objects

After all its child view objects have completed their display tasks, the **`IfThenElseStmntView`** object then displays four links and one circle node, as shown in Figure 5, to indicate the control-flow information visually.

Maintaining internal program representations consistently

The program-database manager, which serves as the integration mechanism, provides infrastructure support for *data*, *presentation*, and *control integration*[30] among all tools in the IVPE. Data integration can be achieved easily because all tools access the internal program representations through the program-database manager. Presentation integration means that all front-end tools, which need to interact with the user, have common and uniform user interfaces. This kind of integration can be achieved easily in our IVPE, because these front-end tools constructed using the MVS class hierarchy use the same set of view and shape objects for dealing with editing and display tasks. Control integration concerns the mechanisms by which one tool activates other tools. The communication channels among tools and the program-database manager in our IVPE are *message-driven*, like that proposed in Reiss.[31] Therefore, a tool may be activated by receiving a message from another tool through the program-database manager.

Table III lists a number of sample methods (or service routines) defined in the **`ProgramDatabaseManager`** class for tool communication. These methods enable various tools to coordinate their actions and maintain the consistency of internal program representations. Method **`Register()`** is used to add tool-registration records to the **`RegisteredToolList`** attribute, while **`Deregister()`** is used to remove them. When a tool finishes modifying the internal program representations, it sends a message called **`ChangeFrom()`**, which may contain the modified data as parameters, to the program-database manager. Upon receiving the **`ChangeFrom()`** message, the

Table II. Class interfaces for supporting construction of program trees (partial)

```
class Model : public TMVObject {
public:
 …
 Model *pParentModel;
 CObList ChildModelList;
 // contains a list of Model objects
 View *pCurrentView;
 CObList DependentViewList;
 //contains a list of View objects
};
class IfThenElseStmnt : public
SelectionStmnt {
public:
 …
 StatementModel *pExpr, *pThenStmnt,
 *pElseStmnt;
};
class SimpleStmnt : public StatementModel
{
public:
 …
 CString SourceCode, Comment;
};
 class Stmnt : public StatementModel {
 …  // pCurrentView = new StmntView(…);
};
class Expression : public SimpleStmnt {
 …  // pCurrentView = new ExpressionView
 (…);
};
```

```
class View : public TMVObject {
public:
 …
 Model *pModel;
 View *pParentView;
};
class IfThenElseStmntView : public
 SelectionStmntView {
public
 …
 StatementView *pExprView,
 *pThenStmntView, *pElseStmntView;
 Circle *pFanInNode;
 SideToTopLink *pToThenStmntLink,
 *pToElseStmntLink;
 BottomToSideLink *pFromThenStmntLink,
 *pFromElseStmntLink;
};
class SimpleStmntView : public
 StatementView {
public:
 …
 Node *pNode;
};
class StmntView : public StatementView {
public:
 …
 Node *pNode;
 // pNode = new  Rectangle(…);
};
class ExpressionView : public
 SimpleStmntView {
 …  // pNode = new Diamond(…);
};
// other model and view classes …
```

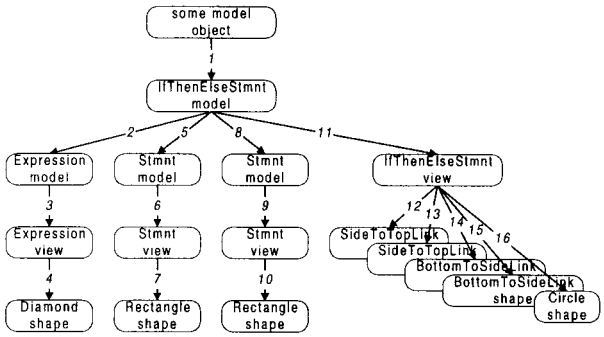

Figure 6. Object creation sequence when inserting an if-then-else statement

Table III. Class interfaces for supporting tool communication and coordination (partial)

```
class ProgramDatabaseManager : public
 CObject {
public:
 …
 CObList RegisteredToolList;
 // contains a list of CObject objects
 …
 int Register(CObject *pTool);
 int Deregister(CObject *pTool);
 int ChangeFrom(CObject *pFromTool, …);
}:
```

```
class FlowBasedEditor : public CScrollView
{
Public=
 …
 int UpdateFrom(CObject *pFromTool, …);
};
class LanguageBasedTextEditor : public
 CScrollView {
public:
 …
 int UpdateFrom(CObject *pFromTool, …);
};
```

program-database manager broadcasts a message called **`UpdateFrom()`** to the tools registered in attribute **`RegisteredToolList`** to retrieve the modified data for further processing.

EDITING AND DISPLAY ACTIVITIES

A language-based editing process

The editing process supported by our editors, the flow-based editor and the language-based text editor, is basically *syntax-directed*. For a placeholder of structured statements, the editors guide the user to replace it with an instance of some structured statement. The above insertion operation is performed when the user selects a valid, i.e. syntactically-correct, template from a template-transformation menu. The locations of graphical templates, including coordinates and dimensions, are calculated automatically by the editors. For example, Figure 7 shows the control-flow graph for the **`ComputerMax`** function before and after the user inserts an if-then statement template into a statement placeholder. For a placeholder of simple statements such as expressions or assignment statements, the editors provide the *in-place* editing (i.e. visual editing) facility, as shown in Figure 8, that helps the user input program text

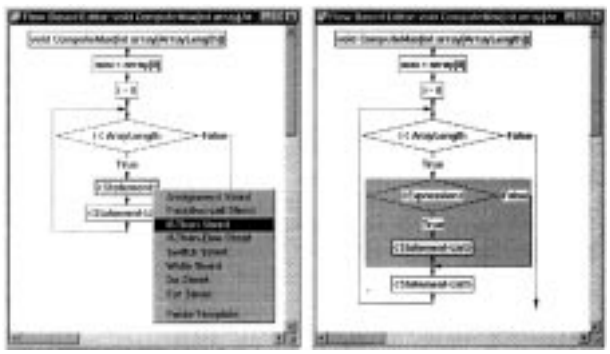

Figure 7. Menu-driven template selection

into the placeholder directly. A parser built into the editors parses and ensures the syntactic correctness of user-input program text.

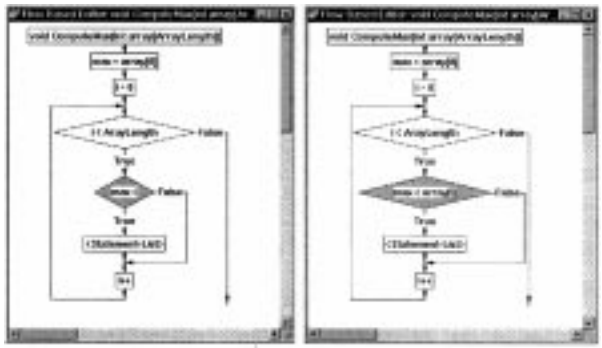

Figure 8. In-place editing

Multi-layered editing facilities

Our IVPE employs two software abstraction techniques called zooming[13,32] and folding[33] to effectively manage software structure and content. Here we only briefly describe related functions, while the implementation details can be found in Hu and Wang.[34]

Module-based layering and the zooming facility

Stepwise refinement, a top-down design methodology for software construction, enables the user to design software by successively refining various levels (e.g. modules or procedures) of implementation details. This kind of software abstraction is called *module-based layering*; that is, the software element to be abstracted corresponds to a module level. Zooming is a well-known graphical interaction facility for handling module-based layering. For example, the **`QuickSort`** function in Figure 9 contains three **`function-call`** statements (i.e. one **`Swap(`**…**`)`** and two **`QuickSort(`**…**`)`** statements) which are depicted as rounded rectangles. When the user issues a zoom-in command on the **`Swap(`**…**`)`** statement, the content of the **`Swap`** function is loaded and displayed on a new window.

Block-based layering and the folding facility

Although existing window-based tools provide the scrolling facility for the user to examine a program page by page, it's sometimes still deficient for the tool to present the whole program structure at one time. A practical IVPE may need to provide an additional abstraction model, called *block-based layering*, that can work on program-statement levels. That is, the user can select and fold (or unfold) blocks of visual or textual program statements on demand. Figure 10 shows such an editing example via the folding facility when the user selects a while statement and issues a reduction command on it. As shown on the right side of Figure 10, the while statement and its constituents are collapsed into an icon, and the associated implemen-

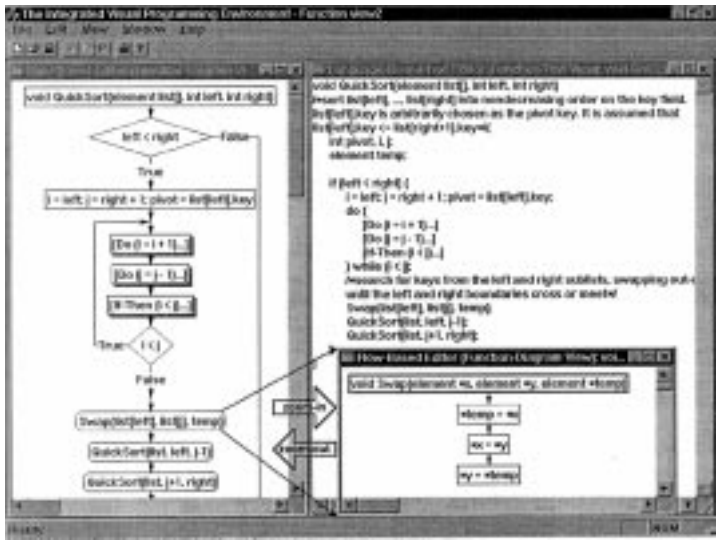

Figure 9. Editing using the zooming facility

tation details are abbreviated. An expansion command, which acts as an inverse operation of reduction, is used to expand the collapsed layout. Figure 11 shows how the folding facility works by enabling a model object to have multiple views dynamically.

TOOL CONSTRUCTION AND INTEGRATION

The MVS class hierarchy presented in this paper is an application framework for facilitating tool construction, including the construction of graphical user interfaces, through compositional reuse of software components. Our construction approach, based on the compositional reuse technique, is to incorporate the tool's functionality into the programming environment by extending the MVS class hierarchy. When a tool is to be introduced, the designer needs to study what functions the new tool will perform, and then examines the MVS class hierarchy to locate attributes and methods in the respective classes that can be reused, as well as new class(es) and associated attributes/methods that need be added. Reusing existing functions, such as those for traversal of program trees, is helpful for reducing the work to construct new tools.

The preceding sections have discussed the kernel of our IVPE, including the program-database manager and the internal program representations that it manages. The following subsections discuss how to construct and integrate a number of programming and analysis tools, including the language-based text editor, the message handler, and the data-flow analyser. The implementation details of the incremental semantic analyser and the program slicer can be found in earlier works.[35,36]

The language-based text editor

Consider the display capacity of a screen; pure text layouts of programs usually take less screen space than graphical layouts. That is, text layouts usually convey

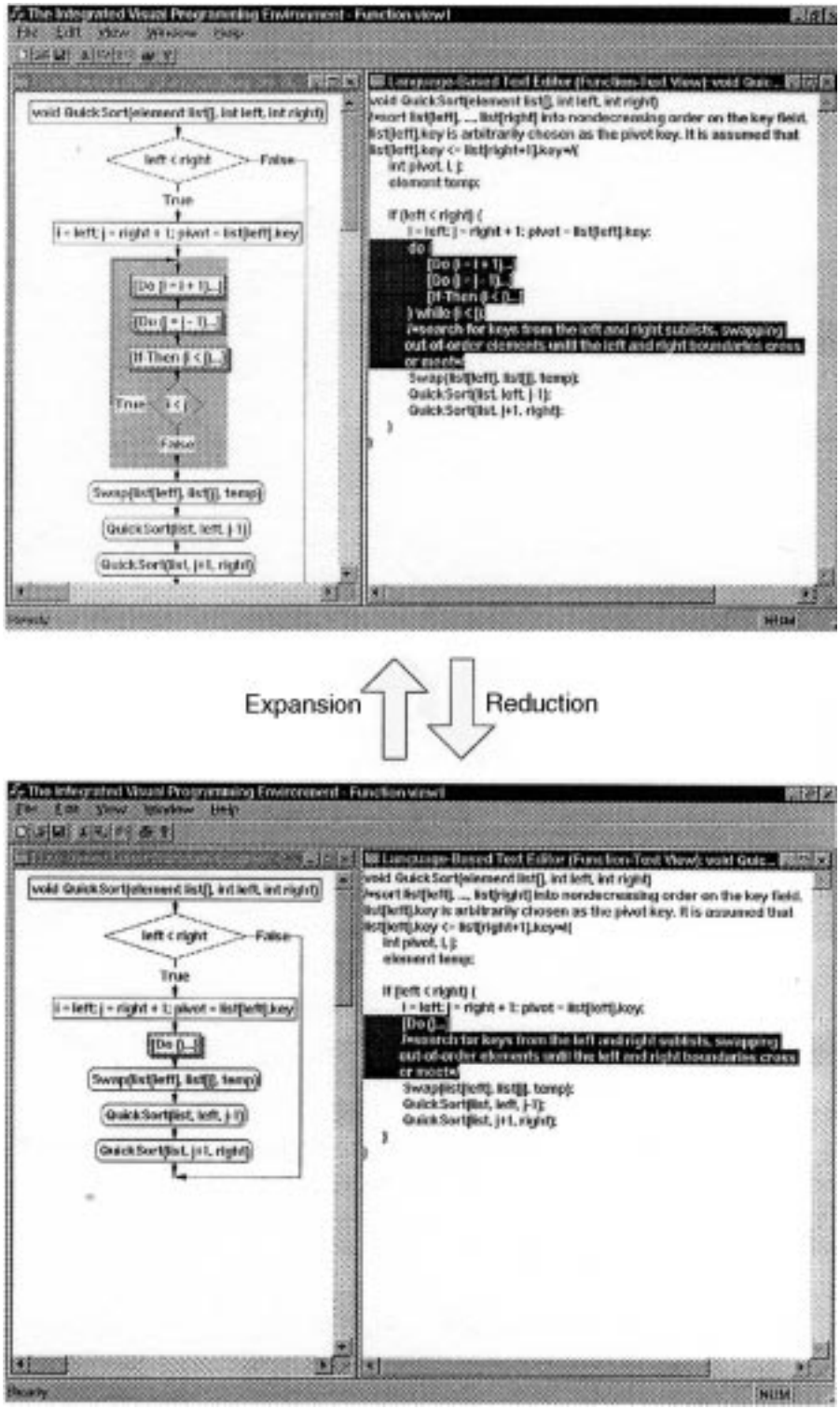

Figure 10. Editing using the folding facility

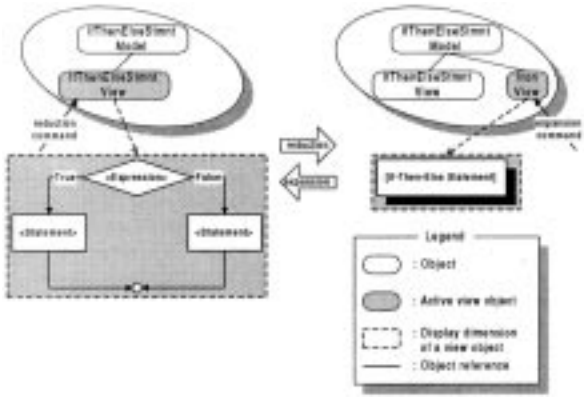

Figure 11. An if-then-else statement model and its two views

more program information than graphical layouts in the same limited display area. To enhance the practicability of the IVPE, a language-based text editor was constructed as an alternate tool for programming. The right side of Figure 12 shows the text layout of the **`ComputeMax`** function as displayed by the language-based text editor. Within the editor, control-flow information about a structured programs is represented implicitly via a sequence of control statements, such as selection and iteration

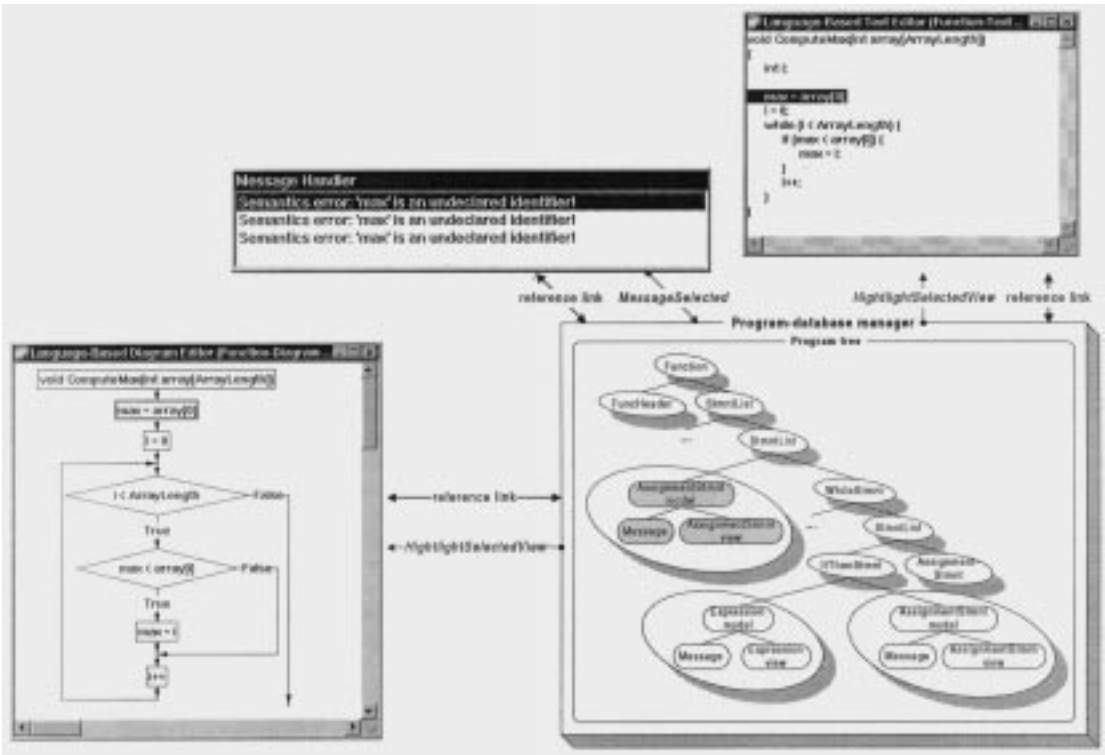

Figure 12. Selecting an error message

statements. By holding language-dependent information, the editor is able to display programs, including source code and comments, in pretty-printed text layouts.

To customize the language-based text editor, the MVS class hierarchy was reexamined and extended as follows:

(a) The **`Model`** class hierarchy. Since the model classes are responsible for managing language-dependent information, all attributes and methods contained in these classes are reused.
(b) The **`View`** class hierarchy. Although the user interfaces of the above two editors look different (i.e. one is graphics-based and the other is text-based), the view-management methods can be reused. The view-presentation methods need be refined for text-only displays.
(c) The **`Shape`** class hierarchy. The **`Link`** class hierarchy is of no use to the language-based text editor. For the **`Node`** class hierarchy, the event-handling methods can be reused, but the graphics-handling methods need be refined.

The message handler

An error or anomaly may be detected during different phases of program analysis (e.g. syntactic, semantic, or data-flow analysis). Here we use a *fault-tolerant* way[12,37] of dealing with these errors and anomalies. That is, an error or anomaly can be tolerated for a certain period of time before it is corrected. In our IVPE, the message handler was constructed to handle error messages. Within the program tree, each model object maintains an attribute called **`Message`** to reference a message object, which also maintains an attribute called **`Model`** to reference the model object. Moreover, each message object maintains an attribute called **`MessageStringList`** to store a list of error messages annotated with error types, such as syntactic or semantic errors. When the program text in a model object includes an error or anomaly detected by the analysis tool, a message object (indicating the occurrence of the error) is created dynamically (by the analysis tool) and attached to the model object. After the analysis tool completes its work, the message handler is invoked (by the program-database manager) to collect message objects by traversing the program tree, and to display these error messages in the *message window*. When the user corrects an error, the analysis tool destroys the corresponding message object(s), so that the error message would disappear from the message window automatically.

For example, as Figure 12 shows, three semantic errors were detected during semantic analysis, and three message objects containing the error messages, **`"Semantic error: 'max' is an undeclared identifier!"`**, were created and attached to the associated model objects. These error messages were then displayed in the message window by the message handler. When an error message is selected in the message window, the message handler sends a message called **`MessageSelected()`** to the program-database manager, which then broadcasts a message called **`HighlightSelectedView()`** to inform the editors to highlight the corresponding program text. When the above semantic error is corrected due to the addition of a variable declaration, the incremental semantic analyser would remove all three message objects.

The data-flow analyser

In the past decade, a number of flow-analysis techniques based on tree manipulation have been explored. *Attribute grammars*[38,39] and *action routines*[11,18,22] are two well-known examples. The common features of these two techniques are that the language semantics is represented as semantic attributes attached to the tree nodes, and flow analysis is performed by traversing the program tree and evaluating the associated attributes' values. Constructing such a flow-analysis technique based on the MVS class hierarchy is straightforward; the functions that a flow analyser performs are implemented by augmenting a number of semantic attributes and evaluation methods to the model class hierarchy.

Our flow-analysis model, like action routines, acts as the *node-marking* process[40] operating on the program tree. The whole analysis is performed via message-passing between model objects in the program tree. When a model object receives a message or gets a return value of the message it sends, it has the best local information to do whatever next action it deems appropriate. That is, the model object may evaluate the attributes' values, send another message to the parent or child model object(s), or just return a specific value. During the flow analysis, those language constructs evaluated as the outcomes are indicated by marking the corresponding model objects. Moreover, the user interface of a new flow analyser does not need to be constructed from scratch because existing view and shape objects, supported in the MVS class hierarchy, can be reused to display the analysis results. This ensures that our IVPE, incorporating a wide range of flow-analysis tasks, provides a uniform and consistent user interface to interact with the user.

The detailed data-flow analysis algorithms, including intraprocedural and interprocedural analyses, based on the message-passing model have been discussed thoroughly elsewhere.[36] Here an example of intraprocedural data-flow analysis is given in this paper. Table IV lists a number of semantic attributes and evaluation methods for computing intraprocedural *definition-use* (DU) and *use-definition* (UD) chains. The semantic attributes which are held by a model class come from two sources: the attributes originally defined in the class; and the attributes inherited from base class(es) of the class. Attributes **`UsedVariables`** and **`DefinedVariables`** are used to store the names of variables that are 'used' and 'defined', respectively. For example, if an assignment statement contains the program text, **`'a`**=**`b`**+**`c', 'b'`** and **`'c'`** will be stored in **`UsedVariables`** and **`'a'`** in **`DefinedVariables`**. Attribute **`Marked`**, a boolean-valued attribute, will be set to **`'TRUE'`** when the model object is included in the analysis results.

In our approach, the functionality of the data-flow analyzer is systematically handled by the following evaluation methods: **`GetUsedVariablesForwardUp()`**, **`GetUsedVariablesForwardDown()`**, **`GetDefinedVariablesBackwardUp()`**, and **`GetDefinedVariablesBackwardDown()`**. The first two methods are responsible for computing DU chains with respect to a variable defined, and the rest for computing UD chains with respect to a variable used. The term 'forward' (or 'backward') shown in the methods' names denotes that the computation sequence would basically follow (or reverse) the control flow of a program. In addition to the above methods, two *activation methods*, **`ComputeDUChain()`** and **`ComputeUDChain()`**, serve as the 'triggers' initiating the DU and UD analyses, respectively.

Figures 13 and 14 show two examples of computing DU chains with respect to

Table IV. Model class interfaces for computing intraprocedural DU and UD chains (partial)

```
class Expression : public SimpleStmnt {
public;
 StringList UsedVariables; // reused
attribute
 …
 void ComputedUDChain(String
variableName,   StatementModel *pFrom,
ModelList
 *pMarkedModels);
 int GetUsedVariablesForwardDown(…);
 /* "…" means that arguments are the same
as  ComputeUDChain() */
};
class AssignmentStmnt : public SimpleStmnt
{
public;
 StringList DefinedVariables,
UsedVariables;
 //  reused attributes
 …
 void ComputeUDChain(…);
 void ComputeDUChain(…);
 int GetUsedVariablesForwardDown(…);
 int GetDefinedVariablesBackwardDown(…);
};
```

```
class SimpleStmnt : public StatementModel
{
public:
 BOOL Marked;
 …
};
class StmntList, IfThenElseStmnt,
WhileStmnt … {
public:
 …
 void GetUsedVariablesForwardUp(…);
 int GetUsedVariablesForwardDown(…);
 void GetDefinedVariablesBackwardUp(…);
 int GetDefinedVariablesBackwardDown(…);
};

/* All internal nodes in the program tree
need to define their respective evaluation
methods for computing DU and UD chains. */
```

variable **a** after the user issued a 'show DU chain' command on the assignment statements **'a=c'** and **'a=b'**, respectively. This command invokes method **ComputeDUChain()** (defined in class **AssignmentStmnt**) to start the DU analysis. The message-passing flow for **GetUsedVariablesForwardUp()** and **GetUsedVariablesForwardDown()** between model objects in the program tree is shown in Figure 15.

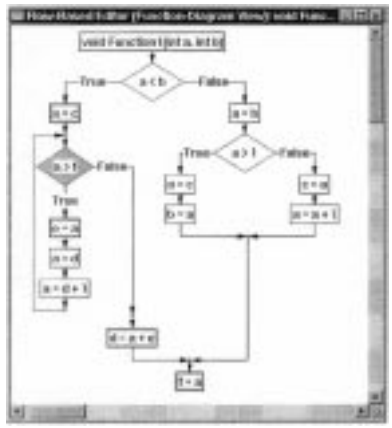

Figure 13. A DU chain w.r.t. variable **a** *in* **'a=c'** *(case 1)*

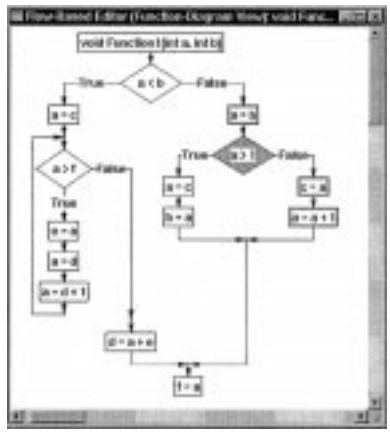

Figure 14. A DU chain w.r.t. variable **a** *in* **'a=b'** *(case 2)*

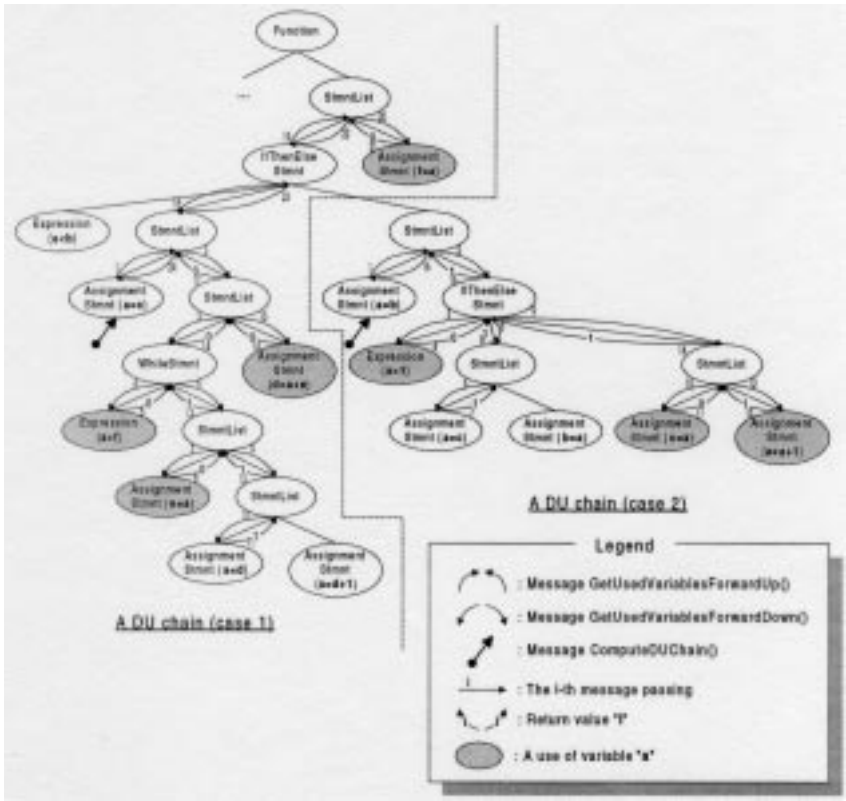

Figure 15. Computing DU chains for Figures 13 and 14

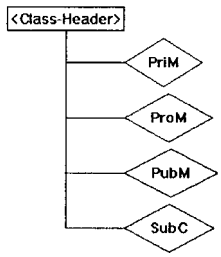

Figure 16. A class template

ENVIRONMENT SUPPORT FOR VISUAL OBJECT-ORIENTED PROGRAMMING

This section briefly discusses how the IVPE can perhaps be used to support more than one language; for example, a C++ language subset. To support visual object-oriented programming in the IVPE, graphical templates for object-oriented language constructs need be designed in advance. Figure 16 shows an example class template, representing the static object-oriented language features such as the class construct and inheritance. After the class template was designed, the next step is to construct the associated model and view classes and add them to the MVS class hierarchy, as shown in Figure 17.

By doing minor modifications of the extended MVS hierarchy, the editors customized are able to support object-oriented program construction and, at the same time, preserve all editing and display facilities mentioned above. The related construction details and descriptions of an IVPE for an object-oriented language can be found elsewhere.[28] Figure 18 shows such a construction example by interacting with the editors. The message handler, a language-independent tool, can be reused without any modification. For those language-dependent analysis tools such as the incremental semantic analyser, their functionalities need be extended (by refining/adding semantic attributes and evaluation methods to the model classes), so that they can work on object-oriented programs.

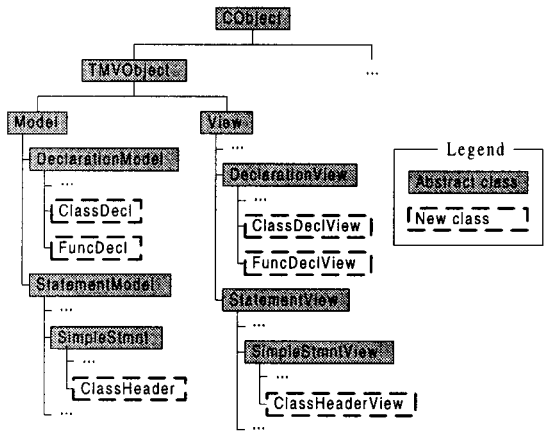

Figure 17. An extended MVS class hierarchy

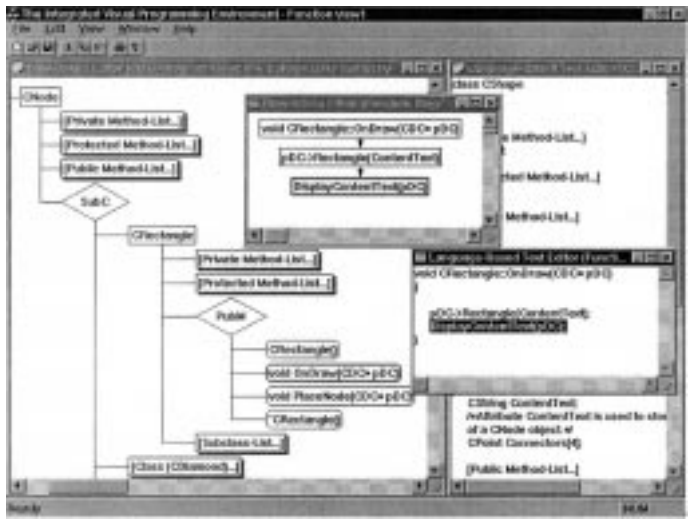

Figure 18. Visual object-oriented programming

RELATED WORK

Prior architectures for interactive applications tend to keep the user-interface part independent from the core application, so that the user-interface part can be reused in constructing different applications. In addition, these architectures realize multiple views by allowing the core application to be attached to several user interfaces. MVC,[16] presentation-abstraction-control (PAC),[41] MVC++,[42] and document-view[43] are such architecture examples. These architectures, which can be viewed as generic design guidelines, may not be useful and intuitive enough to specify the architectural details of some specific applications that handle fine-grained application semantics and a variety of graphical objects. The following gives some comparisons between MVS and MVC architectures.

In the original MVC architecture, an interactive application is divided into three components: model, view, and controller objects. One object-oriented approach[35] to constructing the kernel of an IVPE with the MVC architecture is to design a suite of model, view, and controller classes for each kind of language construct. In this way each node in the program tree corresponds to an MVC triad. For example, Figure 19 shows such a partial program tree representing an if-then-else statement.

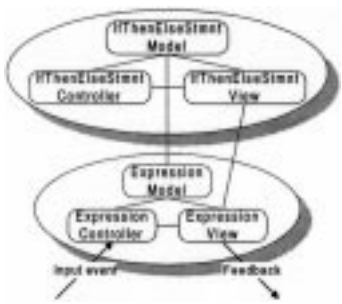

Figure 19. Relationships among model, view, and controller objects

It can be seen from this figure that each object in an MVC triad has an (explicit or implicit) reference to each of the other two objects. View and controller objects are usually tightly coupled and come in pairs, i.e. VC pairs. However, this approach may increase the modification cost of the user-interface part because the implementation details of a view (or controller) class may not be modified alone without considering the partner together. In contrast to the MVC architecture, the MVS architecture enforces a layered and loosely-coupled structure. That is, objects only in 'adjacent layers' are allowed to communicate with each other by invoking predefined protocols. As long as these protocols remain intact, for example, shape objects can be augmented with new functionalities directly without affecting (or being affected by) view objects.

Because an IVPE needs to employ a mass of graphical objects to present the graphical layouts of programs, the functionality of the user-interface part is better classified systematically into two levels: low-level drawing/event-handling facilities and high-level program presentation management, which correspond to shape and view objects in the MVS architecture. This kind of classification may result in more independent and reusable shape classes, and more maintainable and extensible view classes. On the other hand, if the MVC architecture is used, the functions of these two levels would be mixed up in view objects. This may mean that most of the original user-interface code will need to be modified when the IVPE is about to support a new language.

Our previous work[35] was devoted to the construction of a language-based text editor based on the Smalltalk-80 environment. The editor prototype, embedded with a parser and an incremental semantic analyser, provides only primitive functions for handling textual layouts of programs. Visual program construction, however, was not supported in that editor. Moreover, in that work we didn't consider how different interactive tools cooperate together to make up an integrated programming environment. In this paper, design and implementation aspects concerning visual and integration issues are discussed in more detail. This paper also shows how to reduce the effort of constructing such an editor by refining and extending the MVS class hierarchy.

CONCLUSION AND FUTURE WORK

This paper presents an adapted object-oriented architecture, called the model-view-shape architecture, for constructing an IVPE. Application designers who want to construct such programming environments will find that the associated design issues discussed in the paper provide useful design guidance. Although the MVS architecture is a modification of the MVC architecture, it seems more practical to specify the architectural details of an IVPE (or other applications) that needs to interact with users to handle a variety of graphical objects. On the other hand, object-oriented techniques are getting more popular and significant. In this paper, object-oriented techniques are applied to construct the MVS class hierarchy for the IVPE in a systematic manner. To show that the MVS class hierarchy has good extensibility and reusability, the 'tool construction and integration' section gives a number of examples to illustrate how new tools were created and integrated with our IVPE by adding new attributes and/or methods to existing classes.

Flow analysis can be used to facilitate program understanding during the mainte-

nance phase. In some researchers' approaches, well-structured (i.e. syntactically and semantically correct) programs are parsed and translated into the corresponding flow graphs, and flow analysers then traverse these flow graphs to report analysis results to the user. Compared with their approaches, flow analysis presented in this paper is based on the underlying program tree. Our tree-based flow analysis techniques have the following advantages. First, a complex flow-analysis task can be decomposed into manageable subtasks that are handled by passing messages between associated tree nodes. These subtasks may be reused to construct new flow analysers.[36] Second, the flow analyser can directly work on the program tree, without the need to create and maintain redundant data structures, such as *program dependence graphs*.[44] Third, the user can request flow analysis during programming, and the flow analyser can deal with incomplete program fragments incrementally as well as executable programs. It is very helpful for program understanding during the programming as well as the maintenance phase.

In our IVPE, a number of useful editing and display facilities, such as zooming and folding, were designed to enable users to efficiently visualize and construct programs. Although these facilities are customized for our IVPE, the theoretical editing models can actually be applied to the construction of other applications with similar editing requirements. Our current programming methodology with respect to tool construction and integration is still *imperative*, i.e. it needs to specify source code to the MVS class hierarchy manually. Moreover, the reuse technique employed is based on compositional reuse. To gain the benefits of both compositional reuse and generative reuse, we are now applying attribute grammars to specify the functions that new tools perform, and then constructing a code generator to automatically generate source code, based on the MVS class hierarchy, for the tools.

Performance and usability analysis[45] is one important research topic that makes the IVPE more usable and robust. We plan to perform usability analysis on the current IVPE, including the analysis of friendliness and user acceptance, for testers of various programming experience levels. Their comments will be the basis for revising the next version. Moreover, the performance impact of flow analysis for large-sized programs needs further study. So far we have extended the MVS class hierarchy, so that the IVPE is able to support visual programming for object-oriented languages,[28] such as a C++ subset. One of our future projects is to enhance the analysis and maintenance tools in our IVPE in order to work on object-oriented programs. On the other hand, object-oriented language features such as encapsulation, inheritance, and polymorphism make object-oriented programs somewhat uneasy to understand and debug.[46] Thus, we are about to construct a dynamic visualization tool for examining dynamic (i.e. runtime) structures of object-oriented programs.

acknowledgements

This research was supported in part by the National Science Council, under contract number NSC 87-2213-E009-002.

REFERENCES

1. B. A. Myers, 'Taxonomies of visual programming and program visualization', *Journal of Visual Languages and Computing*, **1**(1), 97–123 (January 1990).
2. T. R. G. Green, 'Noddy's guide to visual programming', British Computer Society, Human Computer Interaction Group, Autumn 1995.

3. N. C. Shu (ed.), *Visual Programming*, Van Nostrand Reinhold, 1988.
4. A. Ambler and M. M. Burnett, 'Influence of visual technology on the evolution of language environments', *IEEE Computer*, **22**(10), 9–22 (October 1989).
5. M. M. Burnett, A. Goldberg and T. Lewis (eds.), *Visual Object-Oriented Programming: Concepts and Environments*, Prentice-Hall, 1994.
6. S. K. Chang (ed.), *Principles of Visual Programming Language Systems*, Prentice-Hall, 1990.
7. M. M. Burnett and M. J. Baker, 'A classification system for visual programming languages', *Journal of Visual Languages and Computing*, **5**(3), 287–300 (September 1994).
8. E. Glinert (ed.), *Visual Programming Environments: Applications and Issues*, IEEE CS Press, 1990.
9. E. Glinert (ed.), *Visual Programming Environments: Paradigms and Systems*, IEEE CS Press, 1990.
10. B. Shneiderman, 'Direct manipulation: a step beyond programming languages', *IEEE Computer*, **16**(8), 57–68 (August 1983).
11. T. Tenma *et al*., 'A system for generating language-oriented editors', *IEEE Transactions on Software Engineering*, **SE-14**(8), 1098–1109 (August 1988).
12. R. A. Ballance, S. L. Graham and M. L. Van De Vanter, 'The pan language-based editing system', *ACM Transactions on Software Engineering and Methodology*, **1**(1), 95–127 (January 1992).
13. K. Halewood and M. R. Woodward, 'A uniform graphical view of the program construction process: GRIPSE', *International Journal of Man-Machine Studies*, **38**(5), 805–837 (May 1993).
14. B. Backlund, O. Hagsand and B. Pehrson, 'Generation of visual language-oriented design environments', *Journal of Visual Languages and Computing*, **1**(4), 333–354 (January 1990).
15. G. Costagliola *et al*., 'Automatic generation of visual programming environments', *IEEE Computer*, **28**(3), 56–66 (March 1995).
16. G. E. Krasner and S. T. Rope, 'A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80', *Journal of Object-Oriented Programming*, **1**(3), 26–49 (August/September 1988).
17. T. Teitelbaum and T. Reps, 'The Cornell program synthesizer: a syntax-directed programming environment', *Communications of the ACM*, **24(**9), 563–573 (September 1981).
18. R. Medina-Mora and P. H. Feiler, 'An incremental programming environment', *IEEE Transactions on Software Engineering*, **SE-7**(5), 472–481 (September 1981).
19. J. C. Grundy *et al*., 'Connecting the pieces', in M. M. Burnett, A. Goldberg and T. Lewis (eds.), *Visual Object-Oriented Programming: Concepts and Environments*, Prentice-Hall, 1994, pp. 229–252.
20. T. R. G. Green and M. Petre, 'When visual programs are harder to read than textual programs', in G. C. van der Veer et al. (eds.), *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6* (*6th European Conference on Cognitive Engineering*), 1992.
21. M. Petre, 'Why looking isn't always seeing: readership skills and graphical programming', *Communications of the ACM*, **38**(6), 33–44 (June 1995).
22. S. P. Reiss, 'PECAN: program development systems that support multiple views', *IEEE Transactions on Software Engineering*, **SE-11**(3), 276–285 (March 1985).
23. S. Meyers, 'Representing software systems in multiple-view development environments', *PhD Dissertation*, Department of Computer Science, Brown University, May 1993.
24. W. Citrin, R. Hall and B. Zorn, 'Addressing the scalability problem in visual programming', *Technical report CU-CS-768-95*, Department of Computer Science, University of Colorado, Boulder, 1995.
25. S. Meyers, 'Difficulties in integrating multiview development systems', *IEEE Software*, **8**(1), 49–57 (January 1991).
26. P. C. Wu and F. J. Wang, 'Framework of a multitasking C++ based programming environment MCPE', *Journal of Systems Integration*, 181–203 (February 1992).
27. P. C. Wu and F. J. Wang, 'The evolution of an object-oriented specification for compilers', *Journal of Information Science and Engineering*, **11**(4), 433–452 (November 1995).
28. C. H. Hu and F. J. Wang, 'Towards a practical visual object-oriented programming environment: desirable functionalities and their implementation', revised and submitted to *Journal of Information Science and Engineering*, 1997.
29. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
30. I. Sommerville, *Software Engineering*, 5th ed, Addison-Wesley, 1996.
31. S. P. Reiss, 'Connecting tools using message passing in the Field program development environment', *IEEE Software*, **7**(4), 57–66 (July 1990).
32. J. Welsh, B. Broom and D. Kiong, 'A design rationale for a language-based editor', *Software—Practice and Experience*, **21(**9), 923–947 (September 1991).

33. H. Mossenbock and K. Koskimies, 'Active text for structuring and understanding source code', *Software—Practice and Experience*, **26**(7), 833–850 (July 1996).
34. C. H. Hu and F. J. Wang, 'Implementing multi-layered editing facilities in a flow-based editor', *Proceedings of the 7th Workshop on Object-Oriented Techniques and Applications*, 1996, pp. 388–396.
35. C. H. Hu, F. J. Wang and J. C. Wang, 'Constructing a language-based editor with object-oriented techniques', *Journal of Information Science and Engineering*, **11**(4), 1–25 (November 1995).
36. C. H. Hu and F. J. Wang, 'Incorporating flow analysis into a flow-based editor', *Proceedings of National Computer Symposium*, Taiwan, 1997, pp. D53–D60.
37. R. Bahlke and G. Snelting, 'The PSG system: from formal language definitions to interactive programming environments', *ACM Transactions on Programming Languages and Systems*, **8**(1), 547–576 (October 1986).
38. T. Reps, T. Teitelbaum and A. Demers, 'Incremental context dependent analysis for language based editors', *ACM Transactions on Programming Languages and Systems*, **5**(3), 449–477 (July 1983).
39. T. Reps, *Generating Language-Based Environments*, MIT Press, 1984.
40. A. M. Sloane and J. Holdsworth, 'Beyond traditional program slicing', *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, 1996, pp. 180–186.
41. J. Coutaz, 'Architecture models for interactive software', *Proceedings of the European Conference on Object Oriented Programming*, 1989, pp. 383–399.
42. A. Jaaksi, 'Implementing interactive applications in C++', *Software—Practice and Experience*, **25**(3), 271–289 (March 1995).
43. *Microsoft Foundation Class Library Reference*, Microsoft Press, 1997.
44. J. Ferrante, K. Ottenstein and J. Warren, 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, **9**(5), 319–349 (July 1987).
45. T. R. G. Green and M. Petre, 'Usability analysis of visual programming environments: a 'cognitive dimensions' framework', *Journal of Visual Languages and Computing*, **7**(2), 131–174 (June 1996).
46. W. Citrin, M. Doherty and B. Zorn, 'The design of a completely visual object-oriented programming language', in M. M. Burnett, A. Goldberg and T. Lewis (eds.), *Visual Object-Oriented Programming: Concepts and Environments*, Prentice-Hall, 1994, pp. 67–93.