整合流程分析於流程為基的編輯器*
Incorporating Flow Analysis into a Flow-Based Editor

胡仲華 王豐堅
Chung-Hua Hu Feng-Jian Wang

國立交通大學資訊工程研究所
Institute of Computer Science and Info. Eng.
National Chiao-Tung University
e-mail: (chhu, fjwang)@csie.nctu.edu.tw

朱正忠
William C. Chu

逢甲大學資訊工程研究所
Institute of Information Engineering
Feng-Chia University
e-mail: chu@fcu.edu.tw

摘要

一個實用的流程為基編輯器必需要能夠內含用來計算各種程式流程資訊的流程分析模組，以至於這些分析模組能夠根據使用者的需求以互動或遞增的模式來操作並顯示分析結果。本篇論文提出了一個根基於物件導向技術的方法來整合流程分析模組到現有的編輯器中。我們目前已經使用此方法在視窗的環境中以Visual C++實作出編輯器雛形與兩個流程分析模組-資料流程器與程式切片器。此外，編輯器本身能夠讓使用者以畫圖的方式建構出程式的控制流程圖。

關鍵字：視覺化程式設計，流程為基的編輯器，物件導向技術，C++

Abstract

For a flow-based editor, it can't be practical enough to meet users' demands unless it includes a number of flow analyzers for computing various flow information interactively or incrementally. In this paper, a feasible approach based on object-oriented techniques is presented to smoothly incorporate flow analyzers into a flow-based editor. So far, an editor associated with two flow-analyzer prototypes, a data-flow analyzer and a program slicer, have been implemented using Visual C++ on the Windows environment. The flow-based editor provides an editing environment that enables users to visualize and construct programs by depicting the associated control-flow graphs.

Keywords: visual programming, flow-based editor, object-oriented technique, C++

1. Introduction

A software system can be described in various representations during different development phases. For example, during the design phase, a variety of flow-based diagrams, such as control-flow, data-flow, state-transition, and entity-relationship diagrams are available to present multi-faceted information about the software. One major benefit introduced by these diagrams is that such visual representations facilitate comprehension and maintenance of existing software systems. This benefit will be increasingly significant as visual programming technologies [2] progress.

Two main requirements need be considered in order to design practical flow-based editors. First, a flow-based diagram is usually associated with semantic meaning. To support the depiction of valid (i.e., syntactically- or even semantically-correct) diagrams, a flow-based editor has to embody diagram semantics as its internal knowledge, and uses that knowledge to offer editing guidance to users. Second, a flow-based editor had better include a number of flow analyzers to compute and present various flow information according to users' demands. Incorporating flow analyzers into a flow-based editor is not an easy task. The construction and integration cost may be high if no effective methodology is employed.

In this paper, a series of methods are presented to meet the design requirements mentioned above. First, an adapted object-oriented architecture, called the *model-view-shape* (MVS) architecture [7], is used to decompose general flow-based editors into three main modules with a layered structure. These three modules, composed of model, view, and shape objects correspondingly, are responsible for specifying/handling diagram semantics, managing diagram presentations, and drawing graphical primitives and handling input

* This research was supported by the National Science Council, under contract number NSC 86-2213-E009-019.

events. On the basis of the MVS architecture, we used Visual C++ and the Microsoft foundation class library to develop a flow-based editor as well as an MVS class hierarchy (for a target language) on the Windows environment. Our flow-based editor provides an editing environment that enables users to visualize and construct programs by depicting the associated control-flow graphs.

To provide a flow-based editor with flow-analysis capabilities, this paper presents a feasible yet effective approach on the basis of the class hierarchy developed. In our approach, the functions that a flow analyzer performs are specified via a collection of *semantic attributes* and *evaluation methods* defined in the model class hierarchy. The whole flow analysis is performed via message-passing between model objects in the program tree, and each model object has the best local knowledge to do whatever next action it deems appropriate. That is, when a model object receives a message it may evaluate the values of related semantic attributes, send messages to the parent- and/or child-node model objects, or just return a specific value. So far two simple flow-analyzer prototypes, a data-flow analyzer and a program slicer, have been successfully incorporated into our flow-based editor. These analyzers can be activated interactively on user's demand to work on incomplete programs during the programming process.

2. A Class Hierarchy for a Flow-Based Editor

Fig. 2.1 shows a class hierarchy, which is based on the MVS architecture, for the construction of our flow-based editor. The Model, View, and Shape class hierarchies correspond to model, view, and shape classes, respectively. The following only briefly discusses the functionality of the class hierarchy, and the associated implementation details can be found in [Hu96b].

In our approach, a model and a view classes are constructed for each kind of language construct defined in the target language. These model and view classes are classified into hierarchies based on the functionalities of language constructs. For example, if-then and if-then-else statements are selection statements, which are also kinds of structured statements. The model, a representation of the application domain, contains attributes and operations for maintaining the diagram's state and behavior. Attributes of model classes can be generally classified into two sets: one for the maintenance of internal program representations, and the other for the storage of language-dependent information, such as source code, comments, and *static semantics*. Methods of model classes are used to perform program analysis as well as language-dependent functions, such as setting source code and comments.

The view, which is used to handle the user interface, contains attributes and operations for managing the diagram's display and input-event interactions. Attributes of view classes are used to store high-level presentation information, such as view dimensions, while methods (of view classes) can be classified into two sets: view-management and view-presentation methods. A view object usually consists of a single or a set of shape objects for graphical presentations. These shape objects are application-independent, i.e., they are thought to be reusable graphical elements.

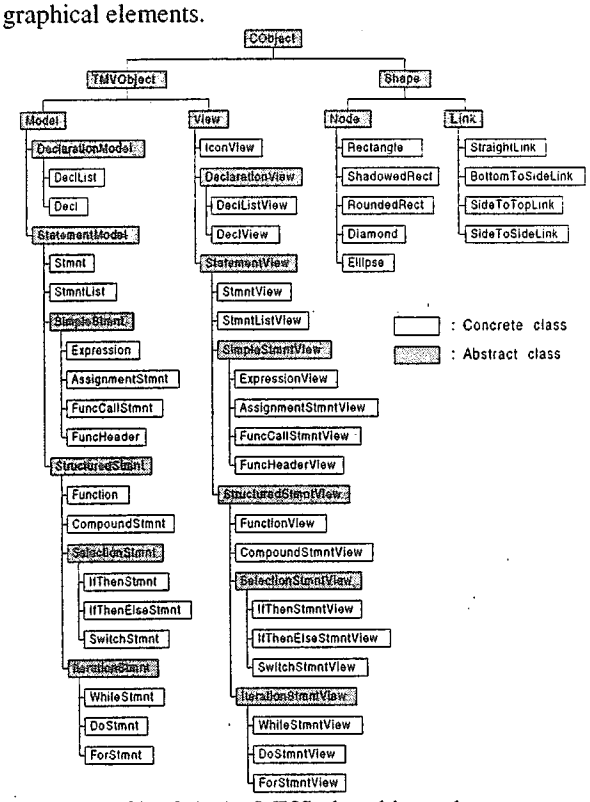


Fig. 2.1: An MVS class hierarchy.

As the user edits (or modifies) a program, a *program tree*, an internal representation of the program, is constructed and maintained incrementally by the flow-based editor. The structure of a program tree is similar to that of an abstract syntax tree; that is, each node in the program tree represents a specific kind of language construct, such as a statement or an expression. Fig. 2.2 shows a sample program tree representing an if-then-else statement, and illustrates association relationships among model, view, and shape objects.

Our current flow-based editor [7] provides flow-based and syntax-directed [9][12] editing facilities that enable users to construct programs by depicting control-flow graphs. For a placeholder of structured statements, the editors guide the user to replace it with an instance of some structured statement. The replacement operation is performed when the user selects a template from a template-transformation menu. The locations of graphical templates, including coordinates and dimensions, are calculated automatically by the editors.

For a placeholder of simple statements, such as expressions or assignment statements, the editors provide the *in-place* editing (i.e., visual editing) facility that helps the user input program text into the placeholder directly. A parser built into the editors parses and ensures the syntactic correctness of user-input program text.

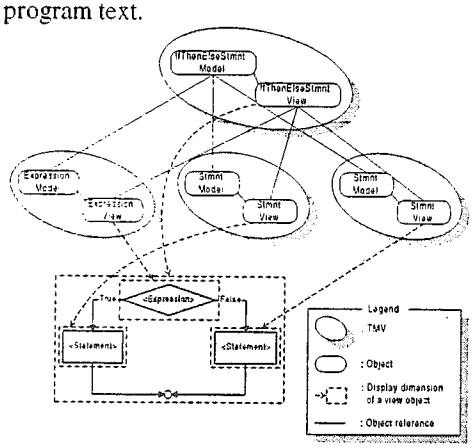


Fig. 2.2: Relationships among model, view, and shape objects.

3. Constructing a Data-Flow Analyzer

3.1 Design Rationales

There are many studies of program flow analysis, such as data-flow analysis [10], ripple-effect analysis [4], and program slicing [13]. These flow analyses are *context-sensitive* because they are closely concerned with underlying language semantics as well as syntax. To provide a flow-based editor with flow-analysis capabilities, the following summarizes three main design issues: 1) How can the language semantics be represented and stored internally in the working memory? 2) What methodology should be employed to evaluate the (internal) semantics for flow analysis? 3) How can the code implemented for current flow analyzers be reused or extended to support the construction of new flow analyzers?

As described in Section 2, a program being constructed is internally represented (by the flow-based editor) as a program tree. In the past decade, a number of flow-analysis techniques based on tree manipulation have been continuously explored. *Attribute grammars* [8] and *action routines* [9] are two well-known examples. The common features of these two techniques are that the program semantics are represented as semantic attributes attached to tree nodes, and the 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. With object-oriented techniques, the functionality of a flow analyzer is implemented by augmenting a number of semantic attributes and evaluation methods to the model class hierarchy, as shown in Fig. 2.1. Reuse of existing attributes and methods reduces the construction cost for new flow analyzers.

Our flow-analysis model, like action routines, acts as the *node-marking* process [11] 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 knowledge to do whatever next action it deems appropriate. That is, a model object may evaluate the attributes' values, send another message to the parent- and/or child-node model object, or just return a specific value. During the flow analysis, those language constructs evaluated to be included in the analysis results are indicated by marking the corresponding model objects. Moreover, the user interfaces of new flow analyzers are of no need to be constructed from scratch because existing view and shape objects, supported in the MVS class hierarchy, can be used to display the analysis results. This entails that our flow-based editor, incorporating a wide range of flow-analysis tasks, provides a uniform and consistent user interface to interact with users.

Most studies employ flow analysis to facilitate program understanding during the maintenance phase. In their approaches, well-structured (i.e., syntactically and semantically correct) programs are parsed and translated into the corresponding flow graphs [5], and flow analyzers then traverse these flow graphs to report analysis results to the user. Compared with them, the flow analysis presented in this paper is based on the underlying program tree. The main benefit of our tree-based analysis is that the user can request flow analysis at will during programming. That is, the flow analysis can deal with incomplete program fragments as well as well-structured programs. This is very helpful for program understanding during programming.

3.2 Intraprocedural Data-Flow Analysis

The data-flow analysis is a process of collecting information about the order that variables are *used* and/or *defined* in a program. *Definition* of a variable x is a statement that assigns a value to x, whereas *use* of a variable x is a statement that references x's value. One of the major tasks performed by (incremental) data-flow analyzers is the computation of data-flow dependencies, such as *definition-use* (DU for short) and *use-definition* (UD for short) *chains* [1], with respect to specific variables. The DU chaining problem is to compute for a definition (i.e., D) of a variable x the set of all the uses (i.e., U) of x, such that there is a control-flow path from D to U which does not redefine x. The UD chaining problem can be informally defined as the computation of a set of the definitions that reach the use of a variable x, i.e., all the statements that directly affect the x's value. The following describes the construction of an interactive data-flow analyzer computing intraprocedural DU and UD chains.

To simplify the discussions of flow analysis based

on the message-passing model, the programs that flow analysis works on are based on structured languages (e.g., C or Pascal) without unconditional jump statements, such as goto statements. Moreover, a number of model classes representing four kinds of typical program statements: simple, sequential, selection, and iteration statements, are considered in this paper. Simple statements, appearing in terminal (i.e., leaf) nodes in the program tree, contain static semantics of variables; for example, the semantic information of variables whose values are used or defined. AssignmentStmnt and Expression model objects, two typical examples of simple statements, act as message initiators and terminators. Sequential, selection, and iteration statements, appearing in nonterminal (i.e., internal) nodes in the program tree, can be mapped to StmntList, IfThenElseStmnt, and WhileStmnt model objects, respectively. These three models objects act as message intermediators. This means that when receiving a message, they may forward it to its parent- and/or child-node model objects according to the their local states.

Table 3.1 lists a number of semantic attributes and evaluation methods for computing intraprocedural DU and 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 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.

Fig. 3.1 shows a complete set of all possible cases (i.e., from Case (1) to Case (12)) in which the model objects, listed in Table 3.1, handle messages received in order to compute DU chains. The following interprets our DU analysis algorithm case by case based on the pure message-passing model. Note that the UD analysis algorithm is not discussed here because it can be deduced using the same object-oriented interpretations.

Table 3.1: Model class interfaces for computing intraprocedural DU and UD chains (partial).

```
class Expression : public SimpleStmnt {
public:
   StringList UsedVariables;
   ......
   void ComputeUDChain(String variableName, StatementModel *pFrom,
                       ModelList *pMarkedModels);
   int GetUsedVariablesForwardDown(...);
   // "..." means that arguments are the same as ComputeUDChain()
};
class AssignmentStmnt : public SimpleStmnt {
public:
   StringList DefinedVariables, UsedVariables;
   ......
   void ComputeUDChain(...);
   void ComputeDUChain(...);
   int GetUsedVariablesForwardDown(...);
   int GetDefinedVariablesBackwardDown(...);
};
class SimpleStmnt : public StatementModel {
public:
   BOOL Marked;
   ......
};
class Function, StmntList, IfThenElseStmnt, WhileStmnt ... {
public:
   ......
   void GetUsedVariablesForwardUp(...);
   int GetUsedVariablesForwardDown(...);
   void GetDefinedVariablesBackwardUp(...);
   int GetDefinedVariablesBackwardDown(...);
};

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

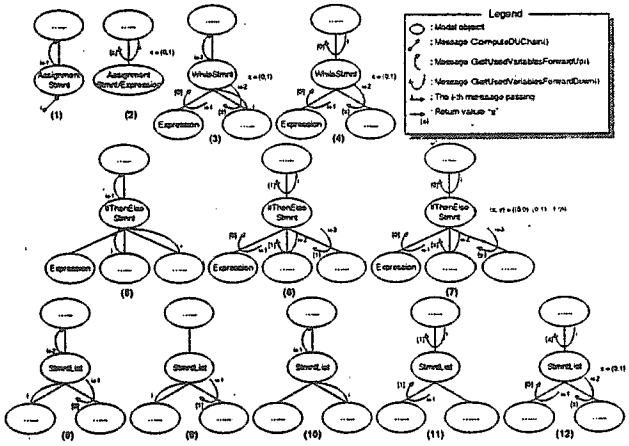


Fig. 3.1: DU analysis scenarios based on the message-passing model.

- AssignmentStmnt model object

  Case (1). When receiving a message ComputeDUChain(), intended to compute a DU chain with respect to a variable (say x) defined in this assignment statement, the AssignmentStmnt object sends a message GetUsedVariablesForwardUp() to its parent-node model object.

- AssignmentStmnt (or Expression) model object

  Case (2). When receiving a message GetUsedVariablesForwardDown(), the AssignmentStmnt (or Expression) object do the following operations if it is the U of x. First, it "marks" itself, and then adds its object reference, e.g., this in C++, to the tail of parameter

pMarkedModels of GetUsedVariablesForwardDown(). In case that the message receiver is an AssignmentStmnt object, it returns a value 1 if it is the D of x. In other cases, the model object returns a value 0.

- StmntList model object

  Cases (8) & (9). When receiving GetUsedVariablesForwardUp() from its left child-node model object, the StmntList object sends GetUsedVariablesForwardDown() to its right child-node model object (say SL), and waits the return value. If the return value reports 0, i.e., Case (8), this means that there exists a control-flow path, originated at SL, which does not redefine x. In this case, StmntList forwards GetUsedVariablesForwardUp() to its parent-node model object to find remaining U's of x. On the other hand, if the return value reports 1, i.e., Case (9), this means that all possible control-flow paths originated at SL redefine x.

  Case (10). When receiving GetUsedVariablesForwardUp() from its right child-node model object, StmntList forwards the same message to its parent-node model object.

  Cases (11) & (12). When receiving GetUsedVariablesForwardDown() from its parent-node model object, StmntList forwards the same message to its left child-node model object, and waits the return value. If the return value reports 1, i.e., Case (11), StmntList also returns a value 1. Otherwise, i.e., Case (12), StmntList forwards GetUsedVariablesForwardDown() to its right child-node model object, waits and returns the return value.

- IfThenElseStmnt model object

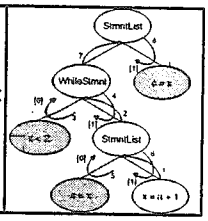
  Case (5). When receiving GetUsedVariablesForwardUp() from its left or right child-node model object (i.e., then or false statement), the IfThenElseStmnt object forwards the same message to its parent-node model object.

  Cases (6) & (7). When receiving GetUsedVariablesForwardDown() from its parent-node model object, IfThenElseStmnt forwards the same message to its child-node Expression object and its rest child-node model objects (say S1 and S2). If the return values replied by S1 and S2 report both 1's, i.e., Case (6), IfThenElseStmnt returns a value 1. Otherwise, i.e., Case (7), IfThenElseStmnt returns a value 0.

- WhileStmnt model object

  Case (3). When receiving GetUsedVariablesForwardUp() from its right child-node model object, the WhileStmnt object sends GetUsedVariablesForwardDown() to both its child-node model objects, and then sends GetUsedVariablesForwardUp() to its parent-node model object. The following shows a message-passing scenario for a sample program fragment to help understand this case.

```
while (x < 2)   -- U of x
{
  a = x;   -- U of x
  x = a+1; -- Compute a DU chain w.r.t. x
}
c = x;   -- U of x
```



  Case (4). When receiving GetUsedVariablesForwardDown() from its parent-node model object, WhileStmnt forwards the same message to both its child-node model objects, and then returns a value 0.

Figs. 3.2 and 3.3 show two examples of computing DU chains with respect to 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 Fig. 3.4.

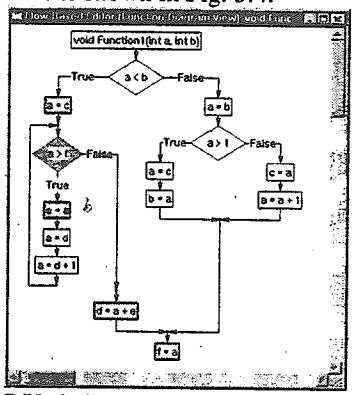


Fig. 3.2: A DU chain w.r.t. variable a in "a=c" (case 1).

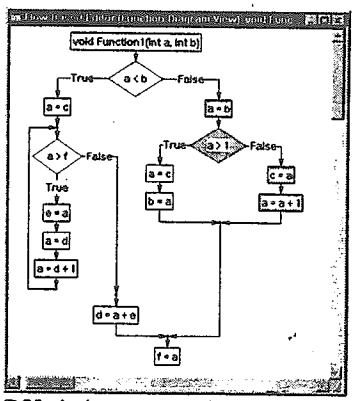


Fig. 3.3: A DU chain w.r.t. variable a in "a=b" (case 2).

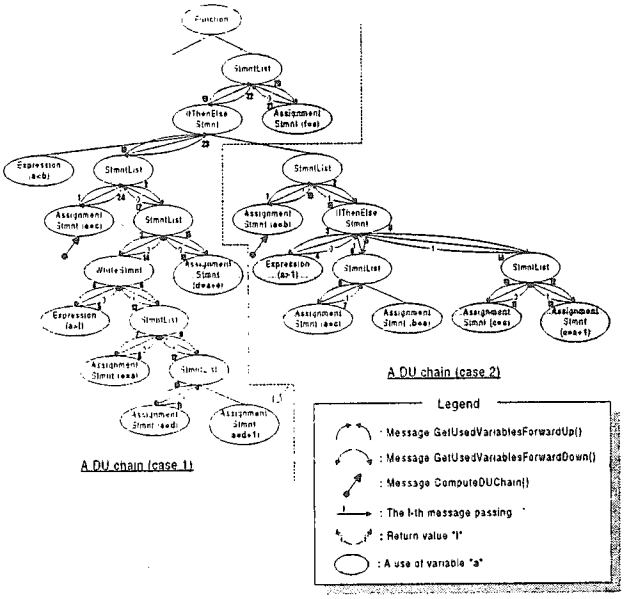


Fig. 3.4: Computing DU chains for Figs. 3.2 and 3.3.

3.3 Interprocedural Data-Flow Analysis

In addition to data-flow analysis within a given function (i.e., intraprocedural analysis), a number of studies concerning interprocedural data-flow analysis can be found in the literature [1][3]. Interprocedural data-flow analysis focuses on computing data-flow information from multiple interacting procedures (or functions). That is, the analysis proceeds across various contexts of caller and callee procedures. For example, as shown in Fig. 3.5, function Function1 invokes function Function2 by passing variables a and b as parameters. In this case the interprocedural DU analysis algorithm need to conclude that those statements in Function2 as well as Function1 are the U's of a (or b) if they directly or indirectly reference a's (or b's) value. The following briefly describes our object-oriented methodology of extending the DU analysis algorithm mentioned above in order to compute interprocedural DU chains. Here we assumed that all parameters in function call are passed by value.

Table 3.2 lists some model class interfaces for computing interprocedural DU chains. Attribute pProgramTree defined in class FuncCallStmnt is used to reference the root-node model object of the callee's program tree, so that the FuncCallStmnt object is able to send message ComputeInterFuncDUChain() to the callee to continue computing the callee's DU chains. Figs. 3.5 and 3.6 show an example of computing an interprocedural DU chain across two distinct functions.

One of the issues in designing an interprocedural DU analysis algorithm is to support recursive function calls and, at the same time, prevent the analysis from being executed infinitely. To tackle this problem, attribute Marked (see Table 3.1) is reused to serve as a checkpoint. When the FuncCallStmnt object sends message ComputeInterFuncDUChain() at the first time, it creates a mark by setting Marked to "TRUE". Afterwards, the FuncCallStmnt object will ignore sending ComputeInterFuncDUChain() if the mark still exists.

Table 3.2: Model class interfaces for computing interprocedural DU chains (partial).

```
class FuncCallStmnt : public SimpleStmnt {
public:
   Function *PprogramTree;
   StringList DefinedVariables, UsedVariables;
   ......
    int GetUsedVariablesForwardDown(String variableName,
              StatementModel *pFrom, ModelList *pMarkedModels);
};
class FuncHeader : public SimpleStmnt {
public:
   StringList DefinedVariables;
   ......
   void ComputeDUChain(String variableName, StatementModel *pFrom,
         ModelList *pMarkedModels
};
class Function : public StructuredStmnt {
public:
   void ComputeInterFuncDUChain(int variableIndex, ModelList *pMarkedModels);
   ......
};
```

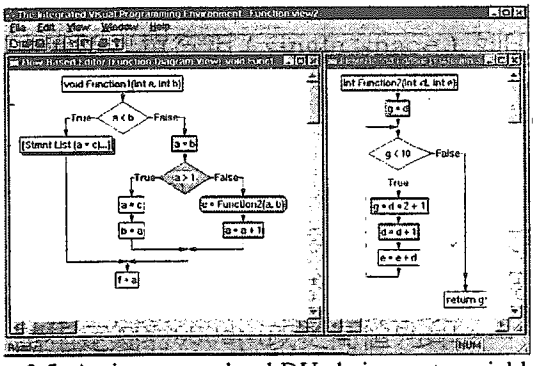


Fig. 3.5: An interprocedural DU chain w.r.t. variable a in "a=b" (Function1).

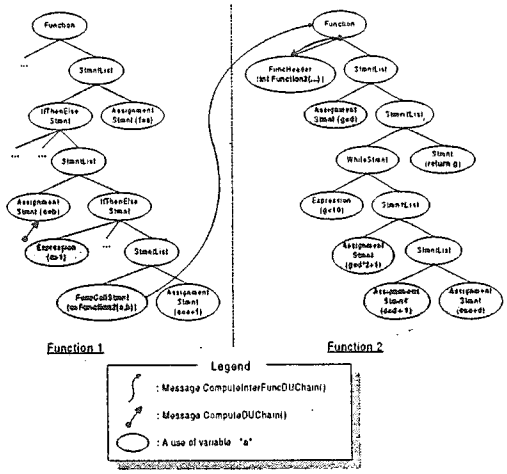


Fig. 3.6: Computing an interprocedural DU chain for Fig. 3.5.

4. Constructing a Program Slicer, another Flow Analyzer

4.1 Design Rationales

Program slicing, an automatic technique determining the statements which may potentially affect (or be affected by) a specific variable at a given

statement, aids program understanding by reducing the amount of code a programmer must examine, and presenting only a relevant program subset of interest. Program slices are generally classified into two categories: *forward slice* and *backward slice*. A forward slice with respect to variable x identifies those statements directly or indirectly affected by x (i.e., referencing x's value), while a backward slice with respect to x identifies those statements that affect x (i.e., assigning a value to x). Program slicing has been applied in many applications, including debugging, testing, maintenance, and reverse engineering.

Computation of program slices involves examining both data-flow and control-flow dependencies of a program. The data-flow analysis has been described in Section 3. On the other hand, statement s is *control dependent* on statement r if r is a predicate (e.g., an expression) that can decide to execute s or not. One typical approach to computing program slices is to summarize and symbolize each control-flow and data-flow dependence as an edge of a directed graph, called the *program dependence graph* [5], in which the vertices are the statements of a program. In this approach, a forward or backward slice is computed by identifying the set of statements in the slice through the forward or backward transitive closure in this graph. However, construction and maintenance of a program dependence graph during programming is not easy due to two major factors. First, the tool designer has to additionally program some, perhaps complicated, data structures (e.g., reference links) to record the dependence relationships among tree nodes. Second, program modification, such as inserting or deleting a statement, would take the editor (much) time to re-calculate the program dependencies in an incremental or a batch way.

The most intuitive yet effective way to construct a program slicer is to reuse data-flow analysis facilities (i.e., the attributes and methods for computing DU and UD chains) and incorporate control-flow analysis facilities into the slicer. The construction cost, compared with the effort based on the building-from-scratch approach, is reasonably low because the tool designer only needs to concern how to reuse the existing code and augment some new functionalities to the MVS class hierarchy. Another advantage is that our program slicer can work directly on the program tree without the need to create and maintain redundant data structures, such as program dependence graphs.

4.2 Intraprocedural Program Slicing

Table 4.1 lists a number of evaluation methods, specified in the respective model classes, for computing intraprocedural program slices. Method ComputeForwardSlice() is used to compute a forward slice with respect to a variable defined, while methods ComputeBackwardSlice(), GetBranchExpressionsBackwardUp(), and GetBranchExpressionsBackwardDown() compute a backward slice with respect to a variable used. GetBranchExpressionsBackwardUp() and GetBranchExpressionsBackwardDown() were designed to track and mark those expressions that potentially affect the execution of a given statement being sliced.

Table 4.1: Model classes interfaces for computing intraprocedural program slices (partial).

```
class Expression : public SimpleStmnt {
public:
   ......
   void ComputeBackwardSlice(String variableName, ModelList *pMarkedModels);
   void GetBranchExpressionsBackwardUp(StatementModel *pFrom,
                  ModelList *pMarkedModels);
};
class AssignmentStmnt : public SimpleStmnt {
public:
   ......
   void ComputeBackwardSlice(String variableName, ModelList *pMarkedModels);
   void ComputerForwardSlice(String variableName,  ModelList *pMarkedModels);
   void GetBranchExpressionsBackwardUp(StatementModel *pFrom,
                  ModelList *pMarkedModels);
};
class StmntList, IfThenElseStmnt, WhileStmnt ..... {
{
public:
   ......
   void GetBranchExpressionsBackwardUp(...);
   void GetBranchExpressionsBackwardDown(...);
};
```

Table 4.2 shows intraprocedural forward and backward slicing algorithms. The forward slicing algorithm, in brief, invokes ComputeDUChain() (i.e., reusing the functionality of the DU analysis algorithm) as a transitive closure way to facilitate the forward slicing process. Likewise, the backward slicing algorithm is highly associated with the functionality of the UD analysis algorithm.

Table 4.2: Intraprocedural forward and backward slicing algorithms.

```
Function AssignmentStmnt::ComputeForwardSlice(variableName, pMarkedModels)
declare
   variableName (IN variable) : the name of a variable that is to be sliced
   pMarkedModels (OUT variable) : a list of model objects constituting
                                  a forward slice
   pModel : a pointer to a SimpleStmnt object
   varName : a variable's name
begin
   ComputeDUChain(variableName, this, pMarkedModels)
/* Initiate a DU analysis w.r.t. variable 'variableName'. After
  ComputeDUChain() completes execution, pMarkedModels will collect
  a list of model objects constituting a DU chain. */
   for each pModel ∈ pMarkedModels do
      if pModel->ObjectType = "AssignmentStmnt" or "FuncCallStmnt" then
         for each varName ∈ pModel->m_DefinedVariables do
            pModel->ComputeDUChain(varName, this, pMarkedModels)
         od
      fi
   od
end
Function Expression::ComputeBackwardSlice(variableName, pMarkedModels)
declare
   variableName (IN variable) : the name of a variable that is to be sliced
   pMarkedModels (OUT variable) : a list of model objects constituting
                                  a backward slice
   pModel : a pointer to a SimpleStmnt object
   varName : a variable's name
begin
   ComputeUDChain(variableName, this, pMarkedModels)
   for each pModel ∈ pMarkedModels do
      for each varName ∈ pModel->m_UsedVariables do
         pModel->ComputeUDChain(varName, this, pMarkedModels)
      od
      pModel->GetBranchExpressionsBackwardUp(NULL, pMarkedModels)
   od
end
```

Fig. 4.1 shows an example of computing a

forward slice with respect to variable a after the user issued a "show forward slice" command on the assignment statement "a=b". Fig. 4.2 shows such a message-passing flow based on the computation of DU chains.

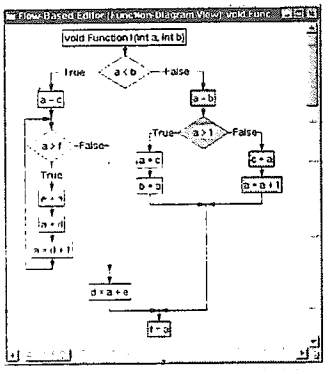


Fig. 4.1: A forward slice w.r.t. variable a in "a=b".

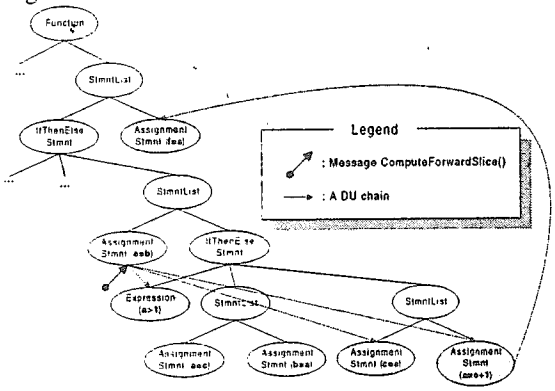


Fig. 4.2: Computing a forward slice for Fig. 4.1.

5. Conclusion and Future Work

Object-oriented techniques, such as inheritance and polymorphism, are getting much popular and significant because they improve the software productivity and quality by promoting software reuse. In this paper, object-oriented techniques are applied to construct a class hierarchy (i.e., the MVS class hierarchy) for the flow-based editor in a systematic manner. Our flow-based editor enables users to construct programs by depicting the associated control-flow graphs. To show good extensibility and reusability the class hierarchy, Sections 3 and 4 give a number of construction examples to illustrate how a flow analyzer is created and integrated with our flow-based editor through the reuse and addition of a collection of semantic attributes and/or evaluation methods.

For the computation and presentation of program flow information, a data-flow analyzer and a program slicer incorporated into the editor provide such an assistance. The functions that these flow analyzers perform can be easily verified through a number of typical structured programs listed in Sections 3 and 4. For a tool designer who wants to construct such a flow analyzer, the design issues discussed in the paper provide useful design guidelines.

Our current program slicer works on the statement level within a function (or procedure). One of our future projects is to extend the functionality of the program slicer by incorporating existing interprocedural slicing algorithms [6], so that the slicer can work on the procedure level (i.e., across multiple functions). On the other hand, object-oriented languages are increasingly popular and important. We plan to extend the data-flow analyzer and the program slicer in order to help understand the static structures of object-oriented programs. The incremental version of these tools will be studied further.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] Ambler, A. and Burnett, M., "Influence of visual technology on the evolution of language environments," *IEEE Computer*, Oct. 1989, pp. 9-22.

[3] Barth, J. M., "A practical interprocedural data flow analysis algorithm," *CACM*, Sep. 1978, pp. 724-736.

[4] Chen, Y. C., *An Efficient Approach to Logical Ripple Effect*, Ph.D. Dissertation, Northwestern University, 1987.

[5] Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Trans. on Prog. Lang. and Sys.*, July 1987, pp. 319-349.

[6] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. on Prog. Lang. and Sys.*, Vol. 12, No. 1, 1990, pp. 26-60.

[7] Hu, C. H. and Wang, F. J., "Constructing flow-based editors with a model-view-shape architecture," *Proceedings of ICS'96*, Taiwan, 1996, pp. 391-397.

[8] Knuth, D. E., "Semantics of context-free languages," *Mathematical Systems Theory*, 1968, pp. 127-145.

[9] Medina-Mora, R. and Feiler, P. H., "An incremental programming environment," *IEEE Trans. on Soft. Eng.*, Sep. 1981, pp. 472-481.

[10] Pollock, L. L. and Soffa, M. L., "An incremental version of iterative data flow analysis," *IEEE Trans. on Soft. Eng.*, Dec. 1989, pp. 1537-1549.

[11] Sloane, A. M. and Holdsworth, J., "Beyond traditional program slicing," *Proc. of the 1996 Intl. Symp. on Software Testing and Analysis*, 1996, pp. 180-186.

[12] Teitelbaum, T. and Reps, T., "The Cornell program synthesizer: a syntax-directed programming environment," *CACM*, Sep. 1981, pp. 563-573.

[13] Weiser, M., "Program slicing," *IEEE Trans. on Soft. Eng.*, July 1984, pp. 352-357.