Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

A Program Slicing System for Object-Oriented Programs

Jien-Tsai Chan and Wuu Yang

Institute of Computer and Information Science
National Chiao-Tung University
Hsinchu, Taiwan, R.O.C.
max@kennedy.cis.nctu.edu.tw and wuuyang@cis.nctu.edu.tw

Abstract

A slice of a program consists of those statements and predicates in that program that may affect, directly or indirectly, the values of certain variables at certain points in the program. Program slicing has been used in program understanding, testing, debugging, software reuse, and program integration. Little has been reported on program optimization by program slicing. We designed and built a program-slicing system as a source code optimizer for object-oriented programs. It can eliminate redundant classes, procedures, variables, and even statements in an object-oriented program.

Key words: program slicing, program dependence graph, program optimization

1. Introduction

Program slicing was introduced by Weiser[Wei84]. He claims that the process of debugging is really a program-slicing action in the programmer's brain. A slice of a program consists of the statements and predicates of the program that may affect the values of certain variables at certain points in the program. Program slicing can identify relevant part of the whole program so that a programmer can focus on a relatively smaller part of the program. This makes debugging and program understanding easier. Furthermore, since a slice of program is an executable part, program slicing can extract reusable components from existing software.

Object-oriented techniques (OOT) are one of the most active research areas in software engineering. It may be viewed as the prelude to the software millennium. OOT supports some abilities to reduce software complexity. While object-oriented programming may lead to cleaner modularity and better component reuse, it also brings about new challenges to software maintainers. A maintainer needs to understand a program before he can modify it correctly. Inheritance, large class libraries and distribution of functionality in object-oriented programs may make programs hard to understand.

In the development of object-oriented programs, there are usually two cooperating teams: the designers of class libraries and the designers of application programs. From a library designer's point of view, a class library should be as versatile and complete as possible. This makes a class library full of all conceivable functions. On the other hand, the application designer wants the compiled code as compact as possible. Unnecessary code may be linked into the compiled program when a large class library is used. A programmer may also write redundant code sometimes. With the program slicing techniques, we can eliminate irrelevant classes, irrelevant functions, unnecessary variables, and even irrelevant statements from the source code of the libarry classes. Program optimization can be accomplished at source code level.

A number of slicing systems have been developed for subsets of the C language, such as Ghinsu [LC92], Unravel[JDJ+95], and Spyder [ADS93]. However, none of these systems can deal with object-oriented programs. Little has been published on the slicing techniques for object-oriented programs. In this paper we apply program-slicing techniques to object-oriented programs and build a practical slicing system. Because existing program representation graphs are not suitable for object-oriented programs, we also suggest a new program representation for object-oriented programs. Besides, a simple and efficient construction method for the representation, an interprocedural data-flow analysis method for OOP, and a modified slicing method are also proposed.

There are many messy problems in building such a system, such as the representation of declarations of variables, functions and classes, parameter passing, control transfer, etc. Several techniques are suggested for these problems in this paper.

The remainder of this paper is organized as follows. The extended program representation for OOP, called OOSDG, is introduced in section 2. Section 3 presents our system structure and implementation techniques, including the

This work was supported in part by National Science Council, Taiwan, R.O.C., under grants NSC 84-2213-E-009-043 and NSC 85-2213-E-009-051.

construction of OOSDG, the interprocedural data-flow analysis method, and the modified slicing algorithm on OOSDG. Finally, section 4 summarizes the paper.

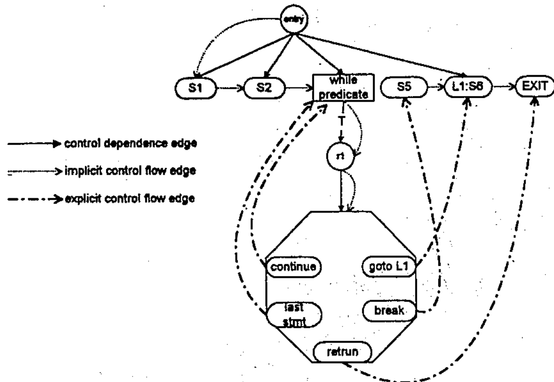

Figure 1. Control depndence graph with implicit and explicit control flow edges.

2. A New Program Representation -- OOSDG

Most program slicing algorithms are based on some kinds of program representation graphs, such as control flow graphs and program dependence graphs(PDG). With a graph representation, program slicing can be defined as a graph-reachablility problem.

Program dependence graphs have received widespread attention of researchers and implementors of language-based tools. A PDG is a graphical representation of a program that contains no procedures. The nodes of the PDG represent the statements, control predicates, and regions of the program. The edges of a PDG mean that there are control- or data-dependence relations between the nodes.

Control dependence is usually defined in terms of a post-dominance relation. For given statements X and Y in a program, we say Y post-dominates X if Y appears in every path from X to the end of program. A statement Y is said to be control dependent on another statement X, if there exists an execution path P from X to Y such that Y post-dominates every node on P, excluding X and Y, but Y does not post-dominate X. It means that there are two execution paths out of X such that one path always leads to the execution of Y and the other may result in Y not being executed. If there is an execution path from X to Y, a variable is defined in statement X and referenced in statement Y, but not redefined along the path from X to Y, Y is said to be data dependent on X.

The original PDG suggested by Ferrante et al. does not encode the control flow information in it [FOW87]. The variant of PDG suggested by Horrald contains explicit control flow edges [HMR93]. Control flow information is encoded in the PDG. The children of a region node are ordered by the order they appear in the program. The order of nodes under the same region node represents the implicit control flow information. Figure 1 shows the control flow information in a PDG. An explicit control flow edge connects the last statement in a *while* loop to the *while* predicate. Control flow edges are also necessary for the transfer statements, such as goto, return, continue and break.

Horwitz et al. extend the original PDG such that their PDG can capture the interprocedural calling contexts and hence it can handle multiple procedures [HRB90]. The extended PDG is called a system dependence graph(SDG). An SDG consists of multiple PDGs, one for each procedure in the program. There are some nodes and edges that are added for handling calling contexts.

Object-oriented programming languages include several mechanisms that do not appear in the traditional programming languages. Conventional program representations, such as PDG and SDG, are not sufficient for OO programming languages because they could not denote the specific information of an OO program, such as inheritance and polymorphism. An extended program representation is required for OO programs. We propose a new program representation for OO programs, called Object-Oriented System Dependency Graph (OOSDG).

Figure 2 depicts the OOSDG of a complete program. The root is a global node that controls the whole program. Global variable declarations, isolated functions (i.e. functions that do not belong to any class), and class declarations are children of the root. The convention in OOSDG is that variable declarations are always placed to the left of other nodes. This order is convenient for data flow analysis.

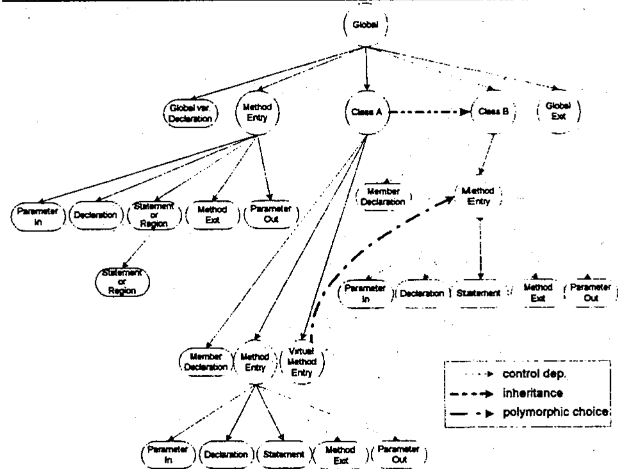

Figure 2. The overview of OOSDG.All the global variables, functions, and classes are control-dependent on the Global node. The data dependence edges and summary edges are omitted.

2.1 The Nodes of OOSDG

Nodes in the CDG represent either simple statements, predicates, or regions of code. The types of statement and predicate nodes are:

- **If Predicate**: The parent of a *then* clause and an *else* clause.
- **While predicate**: A *while* predicate indicates the loop entry. It is also the target of control flow of the continue statements and the last statement in the loop (except the transfer statements).
- **Expression**: A general arithmetic expression.
- **Call**: A simple function call.
- **Declaration**: A variable declaration.
- **Actual-in**: An actual parameter that would be passed to a method.
- **Actual-out**: An actual parameter that receives the return value of the called method.
- **Formal-in**: A formal parameter that receives the value that is passed from the caller.
- **Formal-out**: A formal parameter that would be returned back to the caller. Usually, they are the return values of functions, changed referenced variable, data members, and global variables with side effect.
- **Goto**: The goto statement, with an explicit control flow edge to the target region node of label.
- **Continue**: The continue statement, with an explicit control flow edge to the while predicate node.
- **Return**: The return statement, with an explicit control flow edge to the exit node of the method.
- **Break**: The break statement, with an explicit control flow edge to the statement node following the while loop.

A region node groups the nodes with the same control dependence. The types of region nodes are:

- **Global**: The parent of all classes, global variables and isolated functions.
- **Global exit**: The end of the whole program.
- **Class head**: This node represents a class. It groups all the members and methods of the class. A class head is connected to the base class with an inheritance edge.
- **Method entry**: The entry of a method. This node is connected with the global node, class heads, and call nodes.
- **Method exit**: The end of a method. This node is needed by the return statements. If the method entry node is in a slice, the method exit node will be included in the slice.
- **Virtual method entry**: The entry of a virtual method. This node is connected to the class head node. This node is different from the method entry node in that a virtual function is bound dynamically at run time. All the related virtual functions are linked together via the polymorphic-choice edges.
- **Then clause**: A region node grouping statements in the *then* part of a predicate.
- **Else clause**: A region node grouping statements in the *else* part of a predicate.
- **While body**: A while body groups the statements of the true part of the while predicate.
- **While exit**: The exit point of a while loop.
- **Label**: A label node groups the statement starting from a label till another label, the end of a while loop or the end of a procedure.

2.2 The Edges of OOSDG

The edges are directed lines, starting from one node and ending on another node in the OOSDG. There are nine kinds of edges:

- **Control dependence**: A control dependence edge always starts from a predicate node or region node and ends on either a statement node or a region node.
- **Data dependence:** A data dependence edge starts from a node where a variable is defined and ends on another node where the defined value is referenced.
- **Control flow**: Control flow information is added to the CDG for data flow analysis. The left-to-right order of nodes under the same region preserves the implicit control flow. Beside, there are explicit control flow edges added to the CDG. For example, the continue statements are connected to the beginning of a while-loop; the break statements are connected to the node that follows the while loop; and the return statements are connected to the method exit node.
- **Call**: A call edge starts from a call node and ends on the entry of a method. Besides the simple call nodes, object declaration statements will become a call node too. The reason is that the instantiation of an object needs to call the constructor function.
- **Declaration dependence:** A declaration dependence edge starts from a node where a variable is declared and ends on another node where the variable is defined.
- **Parameter-in**: A parameter-in edge starts from an actual-in parameter node and ends on the corresponding formal-in parameter node.
- **Parameter-out**: A parameter-out edge starts from a formal-out parameter node and ends on the corresponding actual-out parameter node.
- **Polymorphic-choice**: A polymorphic-choice edge connects the entry nodes of virtual functions between subclasses and superclasses. Because the function that is actually called is determined at run time, all possible candidate functions should be included in the slice. With the polymorphic-choice edges and inheritance

edges, all of the relevant virtual functions are included in the slice.

- **Inheritance**: The inheritance edges connect the superclass head to the heads of the inherited subclasses.

2.3 Discussion

A statement may be treated as one node or may be decomposed into several nodes for the abstract-syntax-tree (AST) node [LC92]. Basically, our system considers a statement as a single node but with a few exceptions. Though many variables may be declard in a single statement, in OOSDG, the declaration of each variable is considered as a single node.

There may be many function invocations in one expression, for instance, **a = b + c(x) + d(y,z)**. Functions can also serve as the parameters in another function call, for instance, **p(x, q(w), r(y, z))**. This representation is difficult to process. Consequently, a complicated expression is represented by several nodes so that each node is an expression with at most a function call. Temporary variables will be introduced if necessary. Two examples are shown in Figure 3.

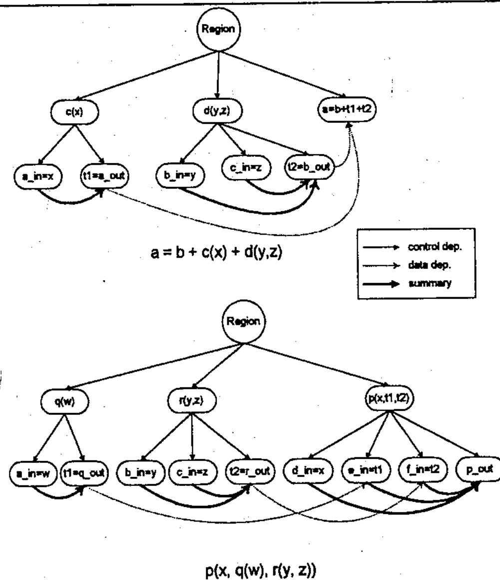

Figure 3. The examples of decomposed expressions. The upper is an example about complicate arithmatic expression. The lower depits the function call as parameters.

The accessing expression of members and methods of an object can be arbitrarily complex, e.g., **a.b(x).c.d.e(y,z)**. As arithmetic expressions and parameters may contain other function calls, the function calls must be separated from the expressions as well. Method invocation is different from a function call in that the method of an object can not be directly extracted out. For example, if a temporary variable **t** is substituted for **b(x)** in the expression **a.b(x).c.d.e(y, z)**, the example will become **a.t.c.d.e(y, z)**. This new expression is wrong because **t** is not a member of object **a**. The correct way is to use a temporary variable to replace the pair of member accessing or method calling sequentially. For this example, first, replace the **a.b(x)** with **t1** and form the new expression **t1.c.d.e(y,z)**. Second, take **t2** to replace the **t1.c** and the expression becomes **t2.d.e(y,z)**. Following this procedure, the original expression will be decomposed into **t1=a.b(x); t2=t1.c; t3=t2.d; t4=t3.e(y,z)** in the order. The resulting nodes are showing in the Figure 4.

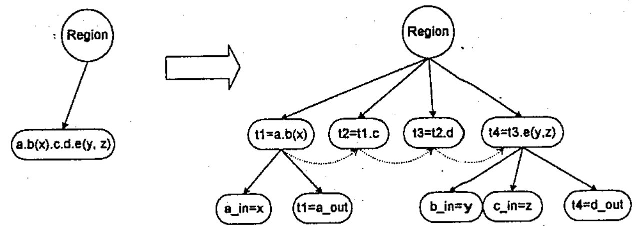

Figure 4. A example of decomposed complicate method call statement.

Members and methods of a class are children of the class head node. Although the members and methods can be placed in an arbitrary order in the original program, the corresponding nodes for members and methods are rearranged in the OOSDG so that all members appear to the left of methods. With this ordering of nodes, data flow analysis is performed in a left-to-right depth-first tree-walk on the control dependence edges.

The head node of a base class and the head nodes of derived classes are connected with inheritance edges. The derived class accesses members and methods of the base class along this edge. All the related polymorphic functions are linked together with polymorphic-choice edges. By these edges, all the possible candidates of a call to a virtual function will be included in the slices.

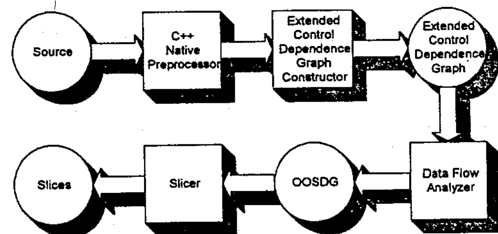

Figure 5. The overview of the program slicing system.

3. System Structure and Implementation Techniques

3.1 System Overview

Figure 5 depicts the overall structure of the slicing system. The system consists of three parts: the extended

control-dependence-graph constructor (parser), the data-flow analyzer, and the slicer. The source program is fed into a C++ native preprocessor first. We use an existing preprocessor to process preprocessor directives. The constructor takes the processed source and constructs an extended control dependence graph (ECDG). After the ECDG is built, data-flow analysis is performed on it. All the interprocedural edges are computed, such as call edges, parameter-in, and summary edges. Finally, the slicer takes the OOSDG and the slicing criteria (designated points and variables of the program) as input and computes the slices. The produced slices are displayed via a user interface.

3.2 Source Preprocessing

C++ is the source language of the slicing system. Since our system does not support C++ preprocessing currently, an ANSI C++ preprocessor is assumed to exist to process all preprocessor's directives. The code produced by the preprocessor is then fed into a standard ANSI C++ parser, which constructs its ECDG.

The constructor can not accept variable length parameter lists currently. If a program contains such functions, the slicing system will not process it.

3.3 OOSDG Construction

3.3.1 Construction Techniques of ECDG

The PDG construction techniques introduced in [CFR+91] and [FOW87] are based on a control flow graph (CFG). CFG is used to identify post-dominators from which control dependencies are computed. These techniques are complicated and difficult to implement. In particular, these methods can not handle the arbitrary jump statements [BH93].

Ballance et al. and Harrold et al. present another technique for constructing CDG [BM92, HMR93]. Their method does not require the CFG and auxiliary data structures. The CDG is constructed while the program is parsed. Then construction is based on abstract syntax trees (AST). The AST is processed in a left-to-right preorder traversal and a corresponding action is used as each node is encountered. This method is easier. The method can handle the structured and unstructured transfers too.

The key to handle the transfer statements is the concept of the follow regions [BM92]. When a block is exited, a new region must be created that will become the current region for statements following the block. The follow region summarizes the control dependence information for the statements following a conditional statement. The concept of the follow regions can be applied to nested conditional statements as well.

Our technique for constructing ECDG is based on Harrold's method [HMR93]. Furthermore, we embed the function for constructing ECDG into the parser. The AST becomes unnecessary. The system works as a one-pass compiler and the target is the ECDG. Every statement translates to the corresponding nodes of ECDG.

3.3.2 Computation of Data Dependence

After the ECDG is constructed, we can perform data-flow analysis on it. There are many kinds of data-flow analyses. All we need to know is the definitions and uses of variables. Because our goal is data dependence edges, we aim at the reaching definitions [FL91][ASU86] that are used to compute flow dependencies for DDG.

A definition is an assignment of a value to a variable in a statement. For a definition of variable V, if there exists a path from the definition of V to a use of V without redefinition of V, we say that the definition of V reaches the use of V. From the reaching definitions, we know the origins where a value may come from. Reaching definitions can be computed with the following equations [FL91]:

$Oub(b) = Def(b) \cup (In(b) - Killed(b))$, b is a node in the control flow graph.

$In(b) = \bigcup_{i \in P(b)} Out(i)$, P(b) is the predecessor of b in the control flow graph.

Four types of data sets are used to compute the set of definitions that reach each statement in the program. All nodes contain IN, OUT, DEF, and USE data sets. The IN set represents the definitions that reach the point immediately before the statement. The IN set of the starting node is the empty set. The OUT set represents the definitions that reach the point immediately after the statement. The DEF set of a statement contains those definitions that reach the end of the node. The variables in the DEF set are those variables that are assigned a value in the statement and global variables and parameters whose values are changed during a call in the definition. All the referenced variables of an expression node are saved in the USE set. The elements of a USE set include the variables on the right-hand side of the assignment, global variables and parameters that are referenced during a function invocation. Data dependence edges can be built with the USE and IN sets. For each variable V defined in the statement, the KILLED set contains all those definitions of V other than the one that appears in the DEF set. The KILLED set clears those definitions overwritten by local definitions in the statement. The KILLED set can be obtained from the intersection of the IN and DEF sets, so it needs not to be saved in a node.

When an object is declared, all the data members of an object are implicitly declared. The declarations of objects add all data members (included those of the ancestor classes) to the DEF set.

```
Procedure: InterproceduralSlicing
Input: an OOSDG and a slicing starting node S in the OOSDG
Output: the slice corresponding to S

begin
    // Phase 1
    MarkReachingNodes(OOSDG, S, {parameter-out})

    // Phase 2
    S' = the marked nodes in OOSDG
    MarkReachingNodes(OOSDG, S', {parameter-in, call})
end

Procedure: MarkReachingNodes
Input: G-->an OOSDG,
       S-->a set of nodes in OOSDG,
       K-->some edge's kinds that will be excepted
begin
    worklist = S
    while worklist is not empty
        select and remove a node V from wordlist
        mark V as in slice
        for each unmarked node w there is a edge w→v whose kind is not in K
            // caputre out the transfer statement exit the procedure, like return
            if w is a method entry node
                add corresponding method exit node to worklist
            else
                if the edge is explicit control flow edge
                    if w is transfer statement(continure,return,break,goto)
                        add w to worklist
                    endif
                else
                    add w to worklist
                endif
            endif
        endfor
    endwhile
end
```

Figure 6. The algorithms for slicing. The procedure InterproceduralSlicing marks the nodes in the slice. The auxiliary procedure MarkReachingNodes marks all nodes in G from which there is a path to a node in V along edges of kinds other than those in the set K.

The DEF set and the USE set are computed and attached to the ECDG node when the ECDG is constructed. Because we are dealing with the ECDG, not the CFG, there are a few things that need to be carefully computed. The IN set is the union of the OUT sets of the preceeding control flow nodes. The preceeding control flow nodes of a node may be found out from the ECDG. These may be the starting nodes of a control dependence edge, an explicit control flow edge, or the nearest left sibling.

The formal-in parameter may not become a formal-out parameter if the formal-in parameter is qualified with a constant qualifier. Except a pointer parameter, the parameter that is passed by value would not be changed. A pointer parameter will involve the aliasing problem. It is an NP-hard problem to find out all the possible aliasing variables, even in the intraprocedural analysis [PLR94]. In our current implementation, we do not find the exact solution and pessimistically treat pointers as *may*-be changed. This new definition does not kill previous definitions but add new definitions to the DEF set. Besides, the parameters that are passed by reference could also be changed.

We developed some techniques to compute the interprocedural reaching definitions [HS94]. Besides the

regular formal-in parameters, all the global variables and the data members within objects are passed as extra parameters. The formal-out nodes represent the return value of a function call, content-changed pointers, changed referenced variables, data members, and global variables with side effect. These variables in the formal-out nodes will be added to the DEF set of the caller node. The referenced extra parameters must be recorded and will be added to the USE set of the caller node.

3.3.3 Summary Edges

Summary edges represent the transitive dependencies between an actual-in node and an actual-out node. The summary edges in an OOSDG serve to circumvent the calling-context problem. Because the summary edges are transitive dependencies and computing slices makes use of the dependencies, we can find out the summary information via intraprocedural slicing. Slicing starts from every formal-out node and traces back to the formal-in nodes via the control dependence and data dependence edges. If it can reach some formal-in nodes, there are summary edges between the starting formal-out node and the formal-in nodes that can be reached. After all summary information for every formal-out node is computed, the information is propagated back to the call nodes via the call edges. Then the actual-in nodes and the actual-out nodes are connected by the summary edges.

3.4 Slicing

A slice of a program with respect to a program point P and a variable V consists of all statements and predicates of the program that might affect the value of V at point P. A slice is a smaller program that reproduces a part of the original program's behavior. Slicing can help a programmer to understand complicated or unfamiliar code.

Our interprocedural slicing algorithm is based on the algorithm suggested by Horwitz et al. [HRB90]. The slicing algorithm consists of a two-pass traversal of the system dependence graph (OOSDG). Assuming the slicing starts at some node S in procedure P. The first phase identifies nodes that can reach S, and are either in P or in a procedure that calls P(either directly or indirectly). The slicing starts from S and goes backward (from target to source) along all the edges in OOSDG except the parameter-out edges. In the first pass, the traversal will not descend into the called procedures. The effects of such procedures are not ignored; they will be captured with the parameter-out edges of actual-out nodes in the second pass.

The second pass identifies nodes that can reach S from procedures called by P (transitively) or from procedures called by procedures that call P (transitively). Slicing starts from nodes that are included in the slice in the first pass and traces backward along all the edges in OOSDG except the call edges and parameter-in edges. The traversal does not ascend into the calling procedures.

The conventional PDG-based slicing algorithms produce incorrect slice when there are unconditional transfer statements such as continue, break, and goto[BH93]. There are two reasons for this problem. First, conventional PDG does not produce nodes for transfer statements. Second, even if the variants of PDG make nodes for the transfer statements, there are no edges flowing out from these nodes. Because slicing is a graph-reachability problem, the nodes representing transfer statements will never appear in any slice. We build nodes for the transfer statements and connect them to the correct targets with control-flow edges. The slicing algorithm is modified for programs with arbitrary transfer statements. If a node is the target of an explicit control flow edge and the source node is a transfer statement, the source node must be included in the slice. The modified algorithm can also handle the inherited class structures and polymorphic methods. The new slicing algorithm is given in Figure 6.

4. Conclusion

In this paper, we propose a suitable program representation (OOSDG) for object-oriented programs. OOSDG can represent the information that comes from new mechanisms of object-oriented programming languages. Inheritance relations that exist between the classes are encoded in OOSDG as well. All the related polymorphic functions are linked together to resolve dynamic binding of virtual functions. Control-flow information is also encoded in OOSDG so that the control-flow graphs are not necessary for data-flow analysis.

Some techniques are developed for constructing OOSDG and for performing data-flow analysis. The construction of OOSDG is embeded in a parser. The concept of the follow regions is introduced to handle the structured and unstructured transfer statements. We use an iterative method to process the interprocedural data-flow analysis. The summary information are also computed with an iterative method.

After translating a program to an OOSDG, a modified slicing algorithm is applied to compute the slices. We have designed and implemented a practical program-slicing system for a subset of C++.

Reference

[ADS93] H. Agrawal, R. A. DeMillo and E. H. Spafford. Debugging with dynamic slicing and backtracking. Software--Practice and Experience, 23(6):589 -616, June 1993.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman. Compilers. Principles, Techniques and Tools. Addison-Wesley, 1986.

[BH93] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In Vol. 749 of Lecture Notes in Computer Science, pp. 206-222. Springer-Verlag, 1993.

[BM92] R. Ballance and B. Maccabe. Program dependence graphs for the rest of us. Technical Report, University of New Mexico, Noverber 1992.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13 (1991) 451-490.

[ES90] M. A. Ellis and B. Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA, 1990.

[FL91] C. N. Fisher and R. J. LeBlanc, Jr. Crafting a compiler. Benjamin/ Cummings, Redwood city, CA, 1991.

[FOW87] J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319-349, 1987.

[HMR93] M. J. Harrold, B. Malloy and G. Rothermel. Efficient construction of program dependence graphs. ACM International Symposium on Software Testing and Analysis, 18(3):160-170, June 1993.

[HRB90] S. Horwitz, T. Reps and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systmes, 12(1):26-60, 1990.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. ACM Transactions on Programming Languages and Systems, 16(2):175-204, 1994

[JDJ+95] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David W. Binkley. Unravel: A CASE Tool to Assist Evaluation of High Integrity Software. Volume 1: Requirements and Design. Technical Report NISTIR 5691. U.S. DEPARTMENT of COMMERCE, Technology Administration National Institute of Standards and Technology, Computer Systems Laoratory, Gaithersburg, MD. 1995

[LC92] P. E. Livadas and S. Croll. System dependence graphs based on parse trees and their use in software maintenance. Technical Report SERC-TR-61-F, University of Florida, Gainesville, 1992.

[PLR94] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural Def-Use associations for C systems with single level pointers. IEEE Transactions on Software Engineering, 20(5):385-403, May 1994.

[Wei84] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, SE-10(4):352-357, July 1984.