Model for Slicing JAVA Programs Hierarchically

Bi-Xin Li[1], Xiao-Cong Fan[2], Jun Pang[3], and Jian-Jun Zhao[4]

[1]*Department of Computer Science and Engineering, Southeast University, Nanjing 210096, P.R. China*

[2]*School of Information Science and Technology, PENNSTATE, University Park, PA 16802, U.S.A.*

[3]*CWI, Kruislaan 413 1098 SJ Amsterdam, Netherlands*

[4]*Department of Computer Science and Engineering, FIT, Fukuoka 811-02, Japan*

E-mail: bx.li@seu.edu.cn; zfan@ist.psu.edu; Jun.Pang@cwi.nl; zhao@cs.fit.ac.jp

Received June 25, 2002; revised January 5, 2004.

Abstract Program slicing can be effectively used to debug, test, analyze, understand and maintain object-oriented software. In this paper, a new slicing model is proposed to slice Java programs based on their inherent hierarchical feature. The main idea of hierarchical slicing is to slice programs in a stepwise way, from package level, to class level, method level, and finally up to statement level. The stepwise slicing algorithm and the related graph reachability algorithms are presented, the architecture of the Java program Analyzing TOol (JATO) based on hierarchical slicing model is provided, the applications and a small case study are also discussed.

Keywords software engineering, hierarchical model, program slicing, JAVA, stepwise algorithm, JATO

1 Introduction

A program slice is actually an abstraction of human's concerns in program understanding[1,2]. Program slicing has been used in a variety of software engineering areas, such as debugging, testing, program comprehension, reverse engineering, software maintenance, software test, software metrics, etc. A slice of a module at statement s with respect to variable v is the set of all statements and predicates that might affect or be affected by the value of v at s. Initially, the algorithm for computing slices is based on data flow analysis. It is suggested in [3] that program dependence graphs (PDGs) could be used to compute slices more efficiently and precisely. An algorithm based on PDGs was presented by Horwitz, Reps, and Binkley[4], where backward slices can be obtained by walking backwards over the PDGs to obtain all the nodes affecting the value of the concerned variables, and forward slices can be obtained by walking forwards over the PDGs to obtain all the nodes affected by concerned variables.

In traditional procedure-based programs, we mainly focus on all kinds of data-flows and control-flows in procedures or among procedures, which can be computed by graph-reachability algorithms[4,5], or two-phase graph-reachability algorithms[3]. In order to slice such kind of procedure-based programs, one must take three steps: 1) construct PDGs, 2) slice PDGs based on graph-reachability or two-phase graph-reachability algorithms and get sliced PDGs, 3) obtain sliced program from sliced PDGs.

However, things get complicated when we take into account object-oriented programs. In addition to considering all kinds of data-flows and control-flows as those in procedure-based programs, we must also consider different dependency relationships originating from the class and object concepts, such as inheritance dependency, message dependency, reference dependency, concurrency dependency, etc.

So far, there are two approaches proposed to slice object-oriented programs. One can be called object-oriented extension that is based on traditional graph-reachability algorithms and two-phase graph-reachability algorithms. Many researchers are doing work along this line. For instance, Krishnaswamy[6] used object-oriented program dependence graph (OPDG), an extension of system dependence graph (SDG), by embedding class hierarchy graphs (CHGs) into SDG. He also proposed an algorithm based on OPDG to slice C++ programs. Larsen and Harrold extended the traditional SDG by adding class dependence graphs (ClDS) in computing C++ program slices[7]. Tip, Choi, Field and Ramalingham used class hierarchy graph (CHG) to compute class hierarchy slices[8]. Steindl presented an approach to computing inter-

*Correspondence

This work is supported by the National Natural Science Foundation of China under Grant No.60473065 and the Outstanding Youth Teacher Support Foundation of Southeast University under Grant No.4009001011.

modular slices in object-oriented program[9]. Zhao proposed a method to compute dynamic object-oriented program slices[10]. All these methods are simply based on some kinds of dependence graphs and graph reachability or two-phase graph reachability algorithms.

The other promising way is to introduce new algorithms to deal with the new object-oriented challenges in slicing object-oriented programs. In this paper, we follow this way and try to go a step further to separate our concerns in program slicing. In other words, we are trying to slice programs in a stepwise way.

The organization of this paper is as follows. An overview of the main idea of our approach is presented in the next section. A hierarchical slicing model (HSM) is introduced in Section 3, where we discuss a hierarchical slicing criterion, hierarchical dependence graphs and a stepwise slicing algorithm. Section 4 introduces the implemented Java program analyzing tool—JATO, which is based on the hierarchical slicing model and the stepwise slicing algorithm. Section 5 makes some comparison with related work, and Section 6 concludes the paper.

2 Basic Ideas

Object-oriented programs have explicit hierarchical structures. Especially, the hierarchical structure in a Java program is very clear, i.e., a package consists of classes and interfaces; a class or an interface consists of methods and member variables; a method consists of some statements and local variables. The HSM constructs dependence graphs and computes program slices over these hierarchical structures to meet the different specific requirements.

The points to stress are as follows. 1) A package, class/interface, or method affects or is affected by variable v if and only if a statement or a predicate in the package, class/interface, or method affects or is affected by the variable v. 2) If a statement or predicate may affect variable v, and the statement or predicate uses variable j, then we say that variable j may affect variable v.

The HSM divides a Java source program into four levels, i.e., package-level, class or interface-level, member method/member variable-level, and statement/local variable-level. A Java program contains some packages, which produce dependencies by *import* statements and sub-package relationships; a package consists of some classes or interface declarations. These classes and interfaces produce dependence relationships between them by inheritance, implement, call to class member, etc. A class or interface declaration consists of some member data and member methods, the connection between methods is realized by calling each other, and a method consists of some statements and variables (including local and global variables).

The main idea of the hierarchical slicing algorithm is: if we consider slicing criterion $\langle s, v\rangle$, where s and v are a statement and a variable in any method, respectively, then we can compute slices w.r.t slicing criterion $\langle s, v\rangle$ according to the following steps using our hierarchical slicing algorithm (in the following steps, $\mathcal{J}$ represents a Java source program, which may be related to some packages):

• Package-level. First we determine all packages related to the package containing s and v based on the direct or indirect dependence relationships caused by **import** statements. We delete all the packages which are not related to the package containing s and v. Finally, we get the package-level slice w.r.t the slicing criterion $\langle s, v\rangle$, marked as $S^1(\mathcal{J})$.

• Class or interface-level. We analyze $S^1(\mathcal{J})$ and delete all classes and interfaces which are not related to the class containing s and v, and get the class or interface-level slice w.r.t the slicing criterion $\langle s, v\rangle$, marked as $S^2(\mathcal{J})$. We have $S^1(\mathcal{J}) \supseteq S^2(\mathcal{J})$.

• Member method or variable-level. We analyze $S^2(\mathcal{J})$ and delete all member methods and variables which are not related to the method containing s and v, and get the member method or variable-level slice w.r.t the slicing criterion $\langle s, v\rangle$, marked as $S^3(\mathcal{J})$. We have $S^1(\mathcal{J}) \supseteq S^2(\mathcal{J}) \supseteq S^3(\mathcal{J})$.

• Statement and local variable-level. We analyze $S^3(\mathcal{J})$ and delete all statements and predicates which are not related to the statement s containing the variable v. We get the statement and local variable-level slice w.r.t the slicing criterion $\langle s, v\rangle$, marked as $S(\mathcal{J})$. We have $S^1(\mathcal{J}) \supseteq S^2(\mathcal{J}) \supseteq S^3(\mathcal{J}) \supseteq S(\mathcal{J})$. By now we get the slice $S(\mathcal{J})$ w.r.t slicing criterion $\langle s, v\rangle$ of a Java program.

3 Hierarchical Slicing Model

The hierarchical slicing model consists of three parts: 1) hierarchical slicing criterion, 2) hierarchical dependence graphs, and 3) stepwise slicing algorithms, which will be covered in this and the next two sections. Fig.1 shows the structure of our hierarchical slicing model. In the HSM, we can obtain the package-level slice (or the class-level slice, or the method-level slice, or the statement-level slice) based on the package-level dependence graph (or the class-level, or the method-level, or the statement-level dependence graph) and the stepwise slicing algorithm.

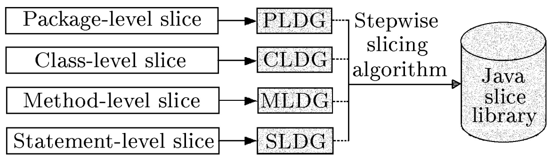


Fig.1. Structure of hierarchical slicing model.

3.1 Hierarchical Slicing Criterion

In this section, we define formally our hierarchical slicing criterion. Given a Java program $\mathcal{J}$, let $P_{\mathcal{J}}$, $C_{\mathcal{J}}$, $M_{\mathcal{J}}$ and $S_{\mathcal{J}}$ represent the set of packages, classes or interfaces, methods(attributes), and statements declared, defined or used in $\mathcal{J}$, respectively.

For $\forall s \in S_{\mathcal{J}}$, we use V_s to denote the variable set occurring (defined or referenced) in statement s. V_s may be empty or a singleton. For any statement $s \in S_{\mathcal{J}}$, we can always get its enclosing method, class or interface, and package by mappings $\mathcal{M}(s)$, $\mathcal{C}(s)$, and $\mathcal{P}(s)$, respectively. For the declaration statement s (or the initializer) of an attribute m, $\mathcal{M}(s)$ is m itself.

The statements that can affect a variable v may be the variable declaration statement, assignment statements with v as the left hand side, or method call statements with v as a call by reference parameter. The statements that can be affected by a variable v may be expressions with one or more occurrences of v, or method call statements with v as a parameter. We denote the set of statements affecting v as *affect*(v), and the set of statements affected by v as *affected-by*(v). Let predicates $B_v = \mathit{affect}(v)$, $F_v = \mathit{affected\text{-}by}(v)$, that is to say, $B_v(s) = \text{true}$ iff $s \in \mathit{affect}(v)$ and $F_v(s) = \text{true}$ iff $s \in \mathit{affected\text{-}by}(v)$.

We say that a package $p \in P_{\mathcal{J}}$, or a class (interface) $c \in C_{\mathcal{J}}$, or a method $m \in M_{\mathcal{J}}$, may affect variable set V if $\exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{P}(s) = p \wedge B_v(s)$, $\exists s \in S_{\mathcal{J}}$, $v \in V \cdot \mathcal{C}(s) = c \wedge B_v(s)$, or $\exists s \in S_{\mathcal{J}}$, $v \in V \cdot \mathcal{M}(s) = m \wedge B_v(s)$, hold.

We say that a package $p \in P_{\mathcal{J}}$, or a class (interface) $c \in C_{\mathcal{J}}$, or a method $m \in M_{\mathcal{J}}$, may be affected by variable set V if $\exists s \in S_{\mathcal{J}}$, $v \in V \cdot \mathcal{P}(s) = p \wedge F_v(s)$, $\exists s \in S_{\mathcal{J}}$, $v \in V \cdot \mathcal{C}(s) = c \wedge F_v(s)$, or $\exists s \in S_{\mathcal{J}}$, $v \in V \cdot \mathcal{M}(s) = m \wedge F_v(s)$, hold.

Thus, all the packages that may affect variable set V are given by $\sum_{P \to V} = \{p | \exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{P}(s) = p \wedge B_v(s)\}$. Similarly, all the classes (interfaces) that may affect variable set V are given by $\sum_{C \to V} = \{c | \exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{C}(s) = c \wedge B_v(s)\}$, and all the methods that may affect variable set V are given by $\sum_{M \to V} = \{m | \exists S \in S_{\mathcal{J}}, v \in V \cdot \mathcal{M}(s) = m \wedge B_v(s)\}$.

All the packages that may be affected by variable set V are given by $\sum_{P \leftarrow V} = \{p | \exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{P}(s) = p \wedge F_v(s)\}$. Similarly, all the classes (interfaces) that may be affected by variable set V are given by $\sum_{C \leftarrow V} = \{c | \exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{C}(s) = c \wedge F_v(s)\}$, and all the methods that may be affected by variable set V are given by $\sum_{M \leftarrow V} = \{m | \exists s \in S_{\mathcal{J}}, v \in V \cdot \mathcal{M}(s) = m \wedge F_v(s)\}$.

Definition 1 (Hierarchical Slicing Criterion). *Given a program $\mathcal{J}$, a slicing criterion with respect to $\mathcal{J}$ is a pair $\langle s, v \rangle$, where $s \in S_{\mathcal{J}}$, $v \in V_s$. By mappings $\mathcal{M}(s)$, $\mathcal{C}(s)$, and $\mathcal{P}(s)$, we can get method-level, class-level and package-level slicing criteria $\langle \mathcal{M}(s), v \rangle$, $\langle \mathcal{C}(s), v \rangle$ and $\langle \mathcal{P}(s), v \rangle$.*

To make any sense, we assume the variable set V_s must not be empty. Thus, in the following we assume $V_s \neq \theta$, for any slicing criterion $\langle s, v \rangle$.

Definition 2 (Package-Level Slice). *Given a program $\mathcal{J}$ and a slicing criterion $\langle \mathcal{P}(s), v \rangle$. The package-level backward slice of $\mathcal{J}$ with respect to $\langle \mathcal{P}(s), v \rangle$ is given by $\sum_{P \to V}$, where $V = \{v\}$; the package-level forward slice of $\mathcal{J}$ with respect to $\langle \mathcal{P}(s), v \rangle$ is given by $\sum_{P \leftarrow V}$, where $V = \{v\}$.*

We can define class-level, method-level and statement-level slices according to the same idea, regarding the slicing criteria $\langle \mathcal{C}(s), v \rangle$, $\langle \mathcal{M}(s), v \rangle$, and $\langle s, v \rangle$, respectively.

3.2 Hierarchical Dependence Graphs

When implementing the hierarchical slicing model, we only consider a subset of Java syntax, where we do not consider the cases including multithreading and exception handling, etc. The hierarchical slicing model consists of three levels: syntax analysis, generation of dependence graphs and computation of slices.

The syntax of a Java source program is analyzed at the syntax analysis level. The code information tree (CIT) and global symbol table (GST) are constructed based on the information obtained by syntax analysis. In the implementation, we first use lex/yacc to produce a CIT. Then all kinds of dependence graphs are generated based on the

CIT produced at the syntax analysis level. Finally, different-level slices can be computed based on these dependence graphs.

Four kinds of dependence graphs are used in the hierarchical slicing model: package level dependence graphs (PLDG), class level dependence graphs (CLDG), method level dependence graphs (MLDG), and statement level dependence graphs (SLDG).

The construction of these dependence graphs depends on the CIT and the GST, which can be obtained by JAST — a built-in special compiler generated by lex/yacc. Logically, the structure of a CIT is equivalent to an abstract syntax tree (AST). Each yacc syntax rule has a corresponding CIT node, which stores all the useful information for later processing. The data structure of the CIT is defined as

```
typedef struct TREENODE
{ struct TREENODE* sibling;
  struct TREENODE* child[4];
  int lineno;      //record line number for slice
  Nodetype nodetype;     //mark node type
  char* name;
  char* datatype;      //datatype including basic and refer-
                         //ence types
  char* casttype;     //record the class name of type cast
                        //between classes
}
```

We can construct a GST by further processing the information involved in a CIT. The GST exhibits the kernel idea of hierarchical slicing model and plays an important role during slice generation. Package-level, class-level and method-level dependence graphs are constructed based on the GST. The GST reorganizes the information above method-level, and divides the information into four levels, i.e., package level, compile unit level, class level and intra-class level, to meet the requirement of later slicing algorithms. The following is the data structure of the GST: a global symbol table is a dynamic array whose element type is class PackageDefine.

```
class PackageDefine //record the information of a package
{ int PackageID; //package identification, begin with zero
  CString PackageName;     //package name
  CStringArray ImportPackages;     //record the informa-
        //tion of all import statements in a package
  CObArray CompileUnitTable;     //record the informa-
        //tion of all compile units in a package, it is a
        //dynamic array whose element type is
        // CompileUnitDefine
}

class CompileUnitDefine      //record the information of a
        //compile unit
{ CString FileName;     //the name of file including com-
        //pile unit
  CObArray ClassTable;     //record the information of
        //all interfaces in compile unit, it is a dynamic
        //array whose element type is ClassDefine
}

class ClassDefine     // record the information of class or
        //interface
{ int ClassID;          //ID of a class or interface
  CString ClassName;     //name of a class or interface
  CStringArray Extends;     //record the information of
        //extends statement in a class or interface
  CStringArray Implements;     // record the information
        //of implements in a class
  CObArray MemberTable;     // record the information
        //of all member variables and member functions in
        //a class or interface, it is a dynamic array whose
        //element type is MemberDefine
}

class MemberDefine     //record the information of all
        //member variables and member functions in a
        //class or interface
{ int MemberID;           //the identification of member
        //variables and member functions
  bool IsMemberData;       //mark is used to determine a
        //member variable or member function
  bool HasMethodBody;     //mark is used to determine a
        //member function or method body
  CString MemberName;     //the name of member variable
        //or member function
  CStringArray DataType;  //the types of all formal para-
        //meters in member function or the types of member
        //variables
  CStringArray CreateClassName;     //record the names
        //of all other classes created in a member function
        //for constructing new relationship
  CUIntArray MemberMethodUseID;  //record ID of all
        //member variables used in a member function for
        //constructing MSV relationship
  CUIntArray MemberDataDefID;       //record ID of all
        //member variables changed in a member function
  CUIntArray MemberDataRefID;       //record ID of all
        //member variables referenced in a member function
  CStringArray MemberDataDef;       //record all varia-
        //bles defined in a member method
  CStringArray MemberDataRef;       //record all varia-
        //bles referenced in a member method
}
```

In next subsections, we introduce the algorithms used to generate the hierarchical dependence graphs. All these algorithms need to traverse some parts of the CIT.

1) **Algorithm 1**: Generation of PLDG. A PLDG is used to represent the *import* and subpackage relationships between packages. If package p_1 imports package p_2, then there is an edge from p_1 to p_2. Because it is the *import* statement that creates the import relationship between packages, so the package-level dependence graph can be constructed directly by scanning CIT.

The data structure of a PLDG is: int* PackageHierarchyDependence. A PLDG is logically a square matrix PA, and the rank of PA is the length

PackageCount of the symbol table *SymbolTable*. The subscripts of *PA* represent the *id* of all packages in source program. $PA[i, j] = 1$ means that the package i imports package j; and $PA[i, j] = 2$ means that the package i is a sub-package of package j.

```
Input: CIT of a program
Output: PLDG
Procedure. Construct·PLDG(CIT *aProgram)
   begin
1. for (int $i = 04$; $i <$ PackageCount; $i$++)
2.   for each PackageDefine pi in SymbolTable
3.     if (pi is a sub-package) then
4.        Create sub-package dependence relationship
5.        Add 2 to corresponding position in the square matrix PA
6.     else
7.        Get the names of all imported packages of pi from pi.ImportPackages
8.        Get the id of each imported package from SymbolTable
9.        Build import dependence relationship between the package and its import package
10.       Add 1 to corresponding position in the square matrix PA
   end
```

For instance, by scanning the CIT we find that there is no import package in the Java program of Fig.2, so its PLDG is the root node itself.

```
1. class A{
2.     public in x;
3.     public f1( )
4.         {x = 1; }
   }
5. class B extends A{
6.     public int y;
7.     public f1( )
8.         {x = 2; }
9.     public f2( )
10.        {y = 3; }
   }
11. class C extends A{
12.     public int y;
13.     public f1( )
14.         {x = y; }
   }
15. class D{
16.     public void f3(A a)
17.         {a.f1( );
18.         return; }
   }
19. class E{
20.     public static void
        main(String str[]){
21.         A a1 = new B( );
22.         D d1 = new D( )
23.         d1.f3(a1); }
   }
```

Fig.2. Toy Java program.

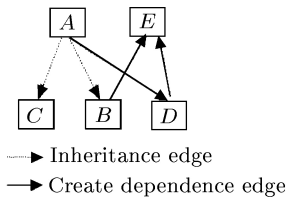
2) **Algorithm 2:** Generation of CLDG. A CLDG is used to represent the inheritance, implement and create relationships between classes or interfaces. If an object of class c_2 is created directly in class c_1, then we say that there is a create relationship between classes c_1 and c_2. If an object of class c_1 is created in class c_2, then there is a bi-direction edge between c_1 and c_2. If class B inherits from class A or implements interface A, then there is an inheritance edge from B to A. Since the inheritance, implement and create relationships in a class or interface are described using extend, implement and new in Java program, the CLDG can also be constructed directly by scanning the CIT.

The data structure of a CLDG is: int* ClassHierarchyDependence. A CLDG is logically a square matrix *CA*. The rank of square matrix *CA* is the number of classes and interfaces in the source program, i.e., *ClassCount*. The subscripts of *CA* represent the id of each class or interface in the source program. The meanings of square matrix are: $CA[i, j] = 1$ denotes that class or interface j inherits class or interface i; $CA[i, j] = 2$ denotes that class j implements interface i; $CA[i, j] = 3$ denotes that class or interface j is a member variable of class or interface i; $CA[i, j] = 4$ denotes that class j creates class i. The algorithm for constructing the CLDG is:

```
Input: CIT of a package
Output: CLDG
Procedure. Construct·CLDG(CIT *aPackage)
   begin
1. for(int $i = 0$; $i <$ ClassCount; $i$++)
2.   foreach ClassDefine ci in SymbolTable
3.   Find the names of all classes inherited by ci in extends statements
4.   Find the names of all interfaces inherited by ci in extends statements
5.   Find the id of each father class or father interface in SymbolTable
6.   Create "inheritance dependence" relationship
7.   Add 1 to corresponding position in square matrix CM
8.   Find the name of each implemented interface by ci
9.   Find the id of each implemented interface in SymbolTable
10.    Create "implement dependence" relationship
11.    Add 2 to corresponding position in square matrix CM
12.    foreach member method or member variable in ci
13.      Get the information inf of member variable or member method in ci
14.    if (inf is a member variable, var with a class type or interface type) then
15.       Find the id of class or interface of the member variable in SymbolTable
16.       Create "aggregation dependence" between ci and class or interface of var
17.       Add 3 to corresponding position in square matrix CM
18.    else (inf is a member method)
19.      Find the name of each created class in CreateClassName
```

```
20.      Find the id of each class in SymbolTable
21.      Create "create dependence" between $ci$
         and the class or interface
22.      Add 4 to corresponding position in square
         matrix $CM$
   end
```

The CLDG of Program given in Fig.2 is shown in Fig.3.



	A	B	C	D	E
A					
B	1				
C	1				
D	3				
E		3	3		

Fig.3. CLDG and its logical matrix of the toy Java program in Fig.2.

3) **Algorithm 3:** Generation of MLDG. In Java programs, class member methods and member variables may have the following two kinds of dependence relationships: 1) multiple member methods share a member variable that leads to a direct definition or use relationship, called MSV (method sharing variable) relationship; 2) call relationship caused by calling other methods so as to use the member variables indirectly. An MLDG is used to describe the two relationships between member methods and member variables.

The data structure of an MLDG is int* MemberDependence. An MLDG is logically a square matrix *MA*. The rank of the square matrix *MA* is the number of all member variables and member methods, i.e., *MemberCount*. The subscripts of *MA* represent the *id* of each member variable or member method in the class. The meanings of the square matrix are: $MA[i,j] = 1$ denotes that member method j uses member variable i (MSV relation); $MA[i,j] = 2$ denotes that member method j calls member method i. The algorithm for constructing the MLDG is:

```
Input: CIT of a class
Output: MLDG
Procedure. Construct_MLDG(CIT *aClass)
   begin
1. for (int $i = 0$; $i < MemberCount$; $i$++)
2. foreach MethodDefine mi in SymbolTable
3.    if (mi is a member method) then
4.       Find the id of each referenced member vari-
         able in mi in MemberDataRefID
5.       Find the id of each referenced member vari-
         able in mi in MemberDataDefID
6.       Find the id of each defined member variable
         in mi in MemberDataRefID
7.       Find the id of each defined member variable
         in mi in MemberDataDefID
8.       Create "MSV dependence" between mi and
         all these member variables
9.       Add 1 to corresponding position in matrix
         MA
10.      Find the id of each member method refer-
         enced in mi in MemberMethodUseID
11.      Create "call dependence" between mi and
         these member methods
12.      Add 2 to corresponding position in matrix
         MA
   end
```

The MLDG of the program given in Fig.2 is shown in Fig.4. The method $f1$ of classes B and C inherited from A may be covered by the method $f1$ newly defined in B and C, respectively. Inherited method $f1$ will not be shown in the MLDG.

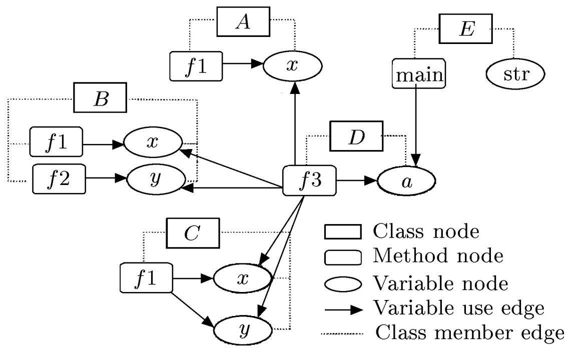


Fig.4. MLDG of the toy Java program in Fig.2.

4) **Algorithm 4:** Generation of SLDG. An SLDG will be derived for each member method of each class. The SLDG of member method m of class c is a pair $\langle N_s, E_s\rangle$, where N_s is the node set, each of the node corresponds to a certain statement in m and E_s is the edge set. There are three kinds of dependence relationships between statements. Statement node j may have data dependence on node i, where variable x is declared and accessible from j. There exists sequential control dependence between a statement and its immediate last statement. And transfer control dependence occurs in conditional statements or loop statements, where the body depends on the predicate part of these statements. E_s is the set which is composed of all these kinds of dependences. The generation algorithm of an SLDG is similar to the traditional algorithms[4,7]. We here do not discuss it in details.

3.3 Stepwise Slicing Algorithm

Based on the hierarchical slicing criterion and different kinds of dependence graph generation algorithms, we can introduce a stepwise slicing algorithm for slicing Java programs. The slicing algorithm includes some continuous steps, like the stepwise refinement method adopted in program development, so we call this algorithm the *stepwise slicing algorithm*.

Firstly, we construct PLDG based on source program, and perform the computation of the package-level slice (in fact, it is a sliced PLDG) over the PLDG w.r.t the package-level slicing criterion $\langle \mathcal{P}(s), v\rangle$.

Secondly, we construct the CLDG based on a sliced PLDG (in fact, it is the class-level extension of the sliced PLDG), and perform the computation of the class-level slice (in fact, it is a sliced CLDG) over the CLDG w.r.t the class-level slicing criterion $\langle \mathcal{C}(s), v\rangle$.

Along this way, we can further obtain the method-level slice and the statement-level slice w.r.t slicing criteria $\langle \mathcal{M}(s), v\rangle$ and $\langle s, v\rangle$, respectively.

In the implementation of slicing algorithm, we define a slicing criterion class *SliceCriterion*.

```
class SliceCriterion
{ set var;          //record the interesting variable or
      //the set of variables
  int lineno;       //record the number of statement s
      //containing varible var
  int packageID;  //record id of the package containing
      //the interesting variable var or the set of
      //variables, we get by mapping P(s)
  int classID;      //record id of the package containing
      //the interesting variable var or the set of var-
      //iables, we get it by mapping C(s)
  int methodID;   //record id of the package containing
      //the interesting variable var or the set of vari-
      //ables, we get it by mapping M(s)
}
```

Then, the algorithm for stepwise program slicing is:

```
Input: Slice criterion ⟨s, v⟩, PLDG
Output: program slice
Procedure. CIT *OnSlicing (PLDG *pa, SliceCriterion *sc)
  begin
  1.  PS=PackageSlicing(pa, sc.packageID)
  2.  CGS={ }
  3.  for all package p in PS do
  4.      CGS=CGS.append(GetCLDG(p))
  5.      CS=ClassSlicing(CGS, sc.classID)
  6.  MGS={ }
  7.  for all class c in CS do
  8.      MGS = MGS.append(getMLDG(c))
  9.      MS=MethodSlicing(MGS, sc.methodID)
  10. SGS={ }
  11. for all method m in MS do
  12.     SGS = SGS.append(getSLDG(m))
  13.     SS=StatementSlicing(SGS, sc.lineno)
  14.     return SS
  end
```

The procedure *PackageSlicing*() is used to get the package level slice of $\langle s, v\rangle$, by taking the PLDG of the system in question and the slicing criterion $\langle \mathit{packageID}, \mathit{var}\rangle$ as parameters, and returning all the packages (or sub-packages) relevant to the package containing statement s. Since all the irrelevant packages are cleared out of the following focus, the overall performance of slicing will be improved considerably.

The procedure *ClassSlicing*() is used to compute the class level slice of $\langle s, v\rangle$, by taking the CLDG (in fact, it is sliced PLDG) and slicing criterion $\langle \mathit{classID}, \mathit{var}\rangle$ as parameters, and returning all the classes or interfaces relevant to the class containing statement s. The slicing focus is further zoomed in to leave all the irrelevant classes or interfaces out of consideration.

The procedure *MethodSlicing*() is used to further determine all the member methods and member variables which are relevant to the method containing statement s, within every relevant class. Likewise, MethodSlicing() returns all the pertinent methods (variables) and ignores the others.

The procedure *StatementSlicing*() is used to compute the statement-level slice of $\langle s, v\rangle$, i.e., the set of all statements and control predicates, which affects the value of variable v for backward slicing (or is affected by the value of variable v for forward slicing). The way we compute the statement level slice is a top-down, instead of a bottom-up, method.

This kind of stepwise slicing approach is based on the hierarchical structure of Java programs, and has been shown in JATO to be an effective and efficient solution to Java program slicing. We will give a case study in the next section. Each of the procedures, *PackageSlicing*(), *ClassSlicing*(), and *MethodSlicing*(), is actually a graph-reachability algorithm. Procedure StatementSlicing() is a two-phase graph-reachability algorithm, which is the same as traditional algorithm[4,7]. For simplicity, here we only give the algorithm for package-level slicing. The others can be done along the same way.

```
Input: package level dependence graph, slicing criterion
Output: package level slicing
Procedure PackageSlicing (PLDG *pa, PNode *node)
  begin
1.  if node is not marked then
2.      mark the node visited
3.      insert node into visitedNodes
4.      for all nodes pred on which node depends do
5.          PackageSlicing(pred);
  end
```

Procedures *GetCLDG*(), *GetMLDG*() and *GetSLDG*() are used to extend sliced PLDG, sliced CLDG and sliced MLDG to class-level, method-level and statement-level dependence graphs, respectively. The concrete processes are given in Algorithms 1–4.

4 JATO—Java Program Analyzing Tool

JATO has been implemented in C++ based on the hierarchical slicing model. JATO can construct different dependence graphs and compute program slices at different levels. The main goal of JATO is to be integrated into a Software Quality Measurement Platform as an embedded system, where program slices of Java programs at different abstraction levels can be computed and used to analyze, measure and test Java programs in the same environment.

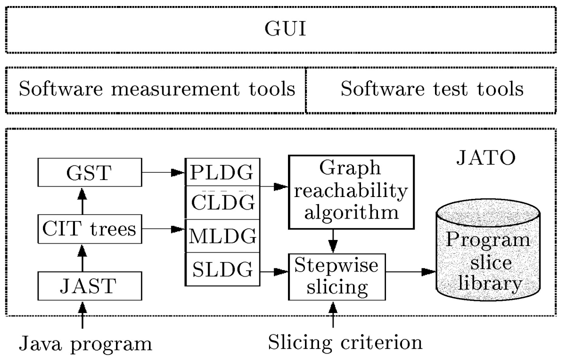


Fig.5. Architecture of JATO.

4.1 Architecture of JATO

The architecture of JATO and its embedding environment are shown in Fig.5. JATO expects Java project files (deployment of packages) as its input. JAST is responsible for generating the abstract syntax trees of the syntactically correct Java programs, and the GSTs are generated by a GST generator. Then, different dependence graphs are derived and fed to the graph-reachability algorithms and the stepwise slicing algorithm to produce the desired program slices.

Measurement and test tools provide some basic tools such as coupling measurement tools, information-flow analysis and test tools, etc. Users can get different-level slices and measurement or test results by interacting with the GUI, which is not discussed in this paper, interested readers are referred to [11, 12] for more information.

4.2 Case Study in JATO

We show the hierarchical slicing approach underpinning JATO by slicing the Java program shown in Fig.2 step by step.

We input a slicing criterion $\langle 23, a1\rangle$, since statement 23 belongs to the *main* method, and belongs to class E. The method level slicing criterion is $\langle main, a1\rangle$, and the class level slicing criterion is $\langle E, a1\rangle$. Assume that all these classes belong to a package named P, then the package level slicing criterion is $\langle P, a1\rangle$. In order to get a slice at the package level, we start from the node that represents package P, find all nodes whose import edges directly or indirectly arrive at node P. The set of all these nodes is the package level slice of the Java program with respect to slicing criterion $\langle P, al\rangle$. In the above example, the package level slice is P itself because only one package is in question. Here, the package level slice (a sliced PLDG) and PLDG are the same, i.e., the P node.

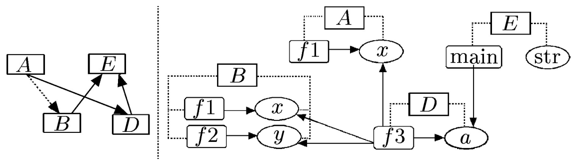


Fig.6. Sliced CLDG and its method-level extension.

In order to get a class level slice, we first extend package node P to the CLDG. In fact, the CLDG can be acquired by traversing the concerned CIT of P and GST. Fig.3 is the CLDG of the example program, where there is an inheritance dependence edge from B to A, as well as from C to A. Since class E creates an object of both class B and class D respectively, there is a create dependence edge from class E to B, as well as from E to D. Then, from the class containing the slicing interested point, statement 23 figures out the set $\mathcal{I}$, which contains all nodes that can be reached from class node E along creation edges, and finds the set $\mathcal{II}$ of all nodes which can be reached from every class node in set $\mathcal{I}$ along inheritance or implement edges. The union of $\mathcal{I}$ and $\mathcal{II}$ is the class level slice with respect to slicing criterion $\langle 23, al\rangle$. Here, $\mathcal{I} = \{B, D, E\}$, $\mathcal{II} = \{A\}$, thus the class level slice w.r.t slicing criterion $\langle 23, al\rangle$ is $\{B, D, A, E\}$. Then, we get the sliced CLDG and its method-level extension in Fig.6.

To get the method level slice, we first extend the sliced CLDG to the MLDG. Since class C is not contained in the class-level slice, we ignore it when we extend the sliced CLDG to the MLDG. Then start to traverse backward from $f3$ node of D along defining edges, and we get node a of D. Then traverse backward from nodes a of D and x of B along used-by edges, and we get nodes $f1$ of B and *main*. Thus, the method level slice is $\{main, D.f3, B.f1, D.a, B.x\}$. To get the statement level slice, we first extend the sliced MLDG to the SLDG along the same way as Fig.6, then we use two-phase graph-reachability algorithm to compute the statement level slice, which is shown in Fig.7.

```
1.  class A{                      17.      {a.f1();
2.      public int x;}            18.       return; }}
5.  class B extends A{            19. class E{
6.      public int y;             20.      public static void
7.      public f1()                        main(String str[]){
8.          {x = 2;}}             21.       A a1 = new B();
15. class D{                      22.      D d1 = new D();
16.     public void f3(A a)       23.      d1.f3(a1);}}
```

Fig.7. Statement slice of Java program in Fig.2 w.r.t. slicing criterion $\langle 23, al\rangle$.

5 Related Work and Comparison

In this section, we make a comparison between our hierarchical slicing model and other object-oriented program slicing approaches, e.g., [6, 7, 13] and consider two aspects: the complexity of the algorithms and the exactness of the slicing results.

5.1 Complexity of the Slicing Algorithm

Compared with the traditional approaches[4–7], the hierarchical slicing model may reduce the whole complexity of the slicing algorithm and the dependence graph construction algorithms, since the different dependence graphs, i.e., PLDG, CLDG, MLDG, and SLDG, can be constructed one upon another. If a slicing criterion is specified, PLDG can be constructed by only considering the packages relevant to the slicing criterion, and leaving the other irrelevant packages out of focus. Thus, the construction time of PLDG w.r.t this slicing criterion will decrease. For the same reason, the construction time of CLDG, MLDG, and SLDG w.r.t this slicing criterion will decrease considerably. However, this is under the assumption that the slicing criterion specified by users is unchangeable, or at least is not changed so frequently. If there are lots of slicing criteria to be specified, the overall performance will get worse since every time a slicing criterion is input, all the dependence graphs, which depend on the slicing criterion, must be re-constructed accordingly.

To overcome this, we can construct these dependence graphs completely, once and for all, for every input Java program. Such pre-computed results can be reused for all slicing criteria selected from the source program. In this way, the construction of SLDG will have the same complexity as that of the traditional approaches. But, the complexity of the slicing algorithm is getting reduced because many of irrelevant packages, classes and methods are ignored when we compute statement-level slices using hierarchical slicing algorithm for a slicing criterion. So, the whole complexity is decreased.

In addition, we can get different level slices using our approach, these by-products are very useful for program comprehension, reverse engineering, code reuse, etc.

5.2 Exactness of Slice Results

One of the main concerns of program slicing is the exactness of the slicing results. One can get a piece of program and claim that it is just the slicing result of the given slicing criterion. However, better program understanding can only be derived from more precise slicing results. In the previous work, the solutions to slice programs with polymorphism are very weak, especially for polymorphic object parameters. A method was proposed to slice programs with polymorphism in [13], but the slicing results are not so exact. It might include the methods in class C or A if we compute the slice with respect to slicing criterion $\langle 23, a1\rangle$ by using their method. The hierarchical slicing model proposed in this paper can eliminate irrelevant classes or methods at class slicing level or method slicing level, so, more precise slicing results can be acquired.

6 Conclusion

The concept of class hierarchy slice was introduced in [8]. This idea was further extended in the hierarchical slicing model[14,15], which covers the package level, method level, in addition to the class level and statement level. In this model, program slices can be derived at different levels with different granularities in a stepwise way. In this paper, the hierarchical slicing model is further elaborated and refined.

The application study of the hierarchical slicing model was presented in [11, 12], where we studied the information-flow analysis and coupling measurement using the hierarchical slicing model.

The prototype tool JATO has been implemented based on the hierarchical slicing model to slice Java programs. One of the aims of the JATO project is to measure and to test Java programs by using object-oriented program slicing techniques. The slice results acquired from JATO can be further exploited in tools concerning software metrics, software test, etc. Currently, the slicing results can be generated correctly in the stepwise slicing algorithm within a Java syntax subset.

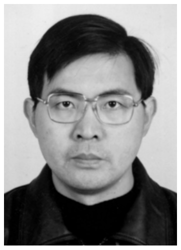
But since the multithreading and exception handling are not included in our algorithm, the wider application of the algorithm is delayed. As a next step, we will extend our algorithm with more Java program features. Some research results show that it is not very difficult to add these properties to our algorithm, but we do not want to discuss these issues in this paper for the simplicity.

Many experiments should be performed in the next stages to improve the functionality of JATO and the software quality measurement platform. Even though we can do some information-flow analysis, coupling assessment and some data-flow test in our environment, there are still many functionalities, for instance functional test, some structural test, cohesion measurement, and code reuse, are not realized in the current version. These will be considered in the next version of the software quality measurement platform.

Acknowledgements The authors would like to thank Yong-Xiang Zhang, Yi Tan, Ping Zhu, Yu-Ting Cheng, Fang Liu, Xiao-Dong Liu, Yu Qian, Zhi-Hong Dong, Liang Jia, Xuan-Dong Li and Guo-Liang Zheng for their contributions in the implementation of the prototype tool—JATO and the Java software quality management platform. Thanks to the reviewers, who made comments and suggestions on the earlier version of this paper, and thanks to Paul Klint and Jϕrgen Iversen for their valuable comments on this paper.

References

[1] Weiser M. Program slicing. *IEEE Transactions on Software Engineering*, 1984, SE-10(4): 352–357.
[2] Weiser M. Program slicing. *IEEE Transactions on Software Engineering*, 1984, 10(4): 352–357.
[3] Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 1984, 19(5): 177–184.
[4] Horwitz S B, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices*, 1988, 23(7): 35–46.
[5] Horwitz S B, Reps T. The use of program dependence graphs in software engineering. In *Proc. the Fourteenth International Conference on Software Engineering*, Melbourne, Australia, May 1992, pp.392–411.
[6] Krishnaswamy A. Program slicing: An application of object-oriented program dependence graphs. *Technical Report TR94-108*, Department of Computer Science, Clemson University, 1994.
[7] Larsen L, Harrold M J. Slicing object-oriented software. In *Proc. 18th International Conference on Software Engineering*, 1996, pp.495–505.
[8] Tip F, Choi J D, Field J, Ramalingham G. Slicing class hierarchies in C++. In *Proc. the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA'96*), 1996, pp.179–197.
[9] Steindl C. Intermodular slicing of object-oriented programs. In *Proc. International Conference on Compiler Construction* (*CC'98*), 1998, pp.264–278.
[10] Zhao J. Dynamic slicing of object-oriented programs. *Technical Report SE-98-119*, Information Processing Society of Japan (IPSJ), 1998.
[11] Li B. An approach for assessing software coupling. In *Proc. Third Asian Workshop on Programming Language and Systems*, Shanghai, China, 2002.
[12] Li B. A technique to analyze information-flow in object-oriented programs. *Information and Software Technology*, 2003, 45(6): 305–314.
[13] Liang D, Harrold M J. Slicing objects using system dependence graph. In *Proc. the 1998 International Conference on Software Maintenance*, 1998, pp.358–367.
[14] Li B, Fan X. JATO: Slicing Java program hierarchically. *TUCS Techinical Reports,* No.416, Finland, 2001.
[15] Li B. Program slicing techniques and its application in object-oriented software metrics and software test [Dissertation]. Nanjing University, 2000.



Bi-Xin Li is a professor in Southeast University from Jan., 2004. He received the Ph.D. degree in computer software and theory from Nanjing University in 2001. From Apr. 2001 to Apr. 2002, he worked at TUCS (Turku Centre for Computer Science) for one year as a post-doctoral researcher. From Apr. 2002 to Dec. 2003, he worked at Department of Computer and Information Science, NTNU (Norwegian University of Science and Technology), and CWI (the Centrum voor Wiskunde en Informatica), both as an ERCIM Fellow. His current research interests include software construction, software testing, SQA techniques, software architecture and component techniques, safety-critical system and formal verification, etc.

Xiao-Cong Fan is a senior researcher in the Intelligent Agent Lab of the Pennsylvania State University from 2002. He received the Ph.D. degree from Nanjing University in 1999. From 2000 to 2002, he worked at the Turku Centre for Computer Science and the Computer Science Department of Abo Akademi University in Finland, where he participated in the projects SOCOS and SPROUT, which developed a methodology for software platform construction based on the Refinement Calculus. He currently works on formal agent theories in teamwork, and projects for applying these theories.

Jun Pang is now a Ph.D. candidate in CWI, the Netherlands. He received the B.Sc. and M.Sc. degrees in computer science from Nanjing University, China, in 1997 and 2000. His research interests include protocol verification, process algebra, safety critical systems, security, testing, software architecture etc.

Jian-Jun Zhao is an associate professor of computer science at Fukuoka Institute of Technology, Japan. He received the B.S. degree in computer science from Tsinghua University, China, in 1987, and the Ph.D. degree in computer science from Kyushu University, Japan, in 1997. His research interests include program analysis and compiler, software architecture analysis, aspect-oriented software development, and ubiquitous computing environment.