Static Slicing of JAVA Programs

Gyula Kovács, Ferenc Magyar, Tibor Gyimóthy

December 1996

TR-96-108

Research **G**roup on **A**rtificial **I**ntelligence
Hungarian Academy of Sciences,
József Attila University
6720, Szeged, Aradi Vértanúk tere 1
HUNGARY
`http://www.inf.u-szeged.hu/~rgai/`

Static Slicing of JAVA Programs

Gyula Kovács
Institute of Informatics, József Attila University
Árpád tér 2, H-6720 Szeged, Hungary
Phone: (36)+(62) 454-145 Fax: (36)+(62) 312-508
e-mail: kovacsgy@inf.u-szeged.hu

Ferenc Magyar
Research Group on Artificial Intelligence
Hungarian Academy of Sciences
Aradi vértanuk tere 1, H-6720 Szeged, Hungary
Phone: (36)+(62) 454-145 Fax: (36)+(62) 312-508
e-mail: magyar@inf.u-szeged.hu

Tibor Gyimóthy
Research Group on Artificial Intelligence
Hungarian Academy of Sciences
Aradi vértanuk tere 1, H-6720 Szeged, Hungary
Phone: (36)+(62) 454-139 Fax: (36)+(62) 312-508
e-mail: gyimi@inf.u-szeged.hu

Abstract

This paper devotes itself to the problem of interprocedural static slicing of JAVA programs. We present a method which focuses on special JAVA features, which can improve the efficiency of a general object-oriented slicing algorithm. In our approach the representation of the hierarchy of JAVA classes is simplified. The method proposed can handle static variables, multiple packages and interfaces. In addition we introduce a new representation in system dependence graphs for polymorphic calls. This approach reduces the number of extra vertices required for handling polymorphism.

1 Introduction

The concept of *program slicing* was originally introduced by Weiser in [Wei84]. In the Weiser concept a slice consists of all statements and predicates that might affect the variables in a set V at a program point p. The pair V and p is defined by a *slicing criterion*, and in some cases [HBR90] it is supposed that the variables in V have to occur at a program point p.

Slicing can be applied to a great number of software engineering tasks e.g. *algorithmic debugging* [Kam93], [FGK91], [PGH94], *incremental* testing [BaH93] and *maintenance* [GaL91]. Slicing techniques can be classified into static and dynamic ones. *Static slicing* is based on an analysis of the program without executing it. A static slice is valid for all executions of the program but it may be imprecise for a particular execution. *Dynamic slicing* involves the program's execution and hence it extracts the precise data flow. However, a dynamic slice must always be produced separately whenever running the program.

A slicing method may produce an executable program or a set of statements. In the first case the generated program computes the same sequence of values for each variable V at program point p. In the second case the slice consists of a set of statements that might influence the value of a variable V at point p.

Intraprocedural slicing algorithms only account for dependences inside a single procedure, while *interprocedural slicing* methods can handle slicing across procedure boundaries.

In [HBR90] Horwitz, Reps and Binkley suggested a graph (called a *system-dependence graph*) for the interprocedural program representation. By using this graph and a two-pass graph reachability

slicing algorithm, precise interprocedural slices can be produced. This slicing algorithm uses summary information at call sites (appearing as summary edges in the graph) to represent the calling context of called procedures. The most time-consuming part of this algorithm is the computation of summary edges. In [RHS94] and [FoG96] methods were suggested for the effective computation of these edges.

In [LaH96] a method was developed for the representation of object-oriented programs with system-dependence graphs (SDGs). In this approach SDGs are created for individual classes, groups of interacting classes, and complete object-oriented programs. On the basis of the created SDGs an efficient interprocedural slicing algorithm can be defined for C++ programs.

We tried to apply the method introduced in [LaH96] for the static slicing of JAVA programs. However, it was soon realised that this method inadequately defines SDGs and slicing algorithms for the full range of JAVA features.

In this paper we develop a method which focuses on the special features of JAVA to achieve a more effective algorithm for the static slicing of JAVA-like programs.

At most, only one *superclass* can be defined for each JAVA class, so a simple tree-like representation is suitable for depicting the hierarchy of JAVA programs, and utilising this approach the size of SDGs can be significantly reduced.

In a JAVA class *static variables* can be defined. These variables can be considered as globals but only for the given class. In C++ programs there are no such variables, hence we introduce a technique for the correct representation of static variables in SDGs.

Our approach is able to produce precise slices for JAVA programs containing several *packages*. The concept of *interface* is then introduced in JAVA. Afterwards we develope a slicing method which generates executable programs from JAVA programs which contain interfaces.

As the representation method of *polymorphic calls* introduced in [LaH96] significantly increases the size of SDGs, we suggest a new method for the implementation of polymorphic calls in SDGs. This approach reduces the number of extra vertices required for handling polymorphisms.

The paper contains a modified version of the algorithm presented in [RHS94] for the computation of summary edges. This modified algorithm is able to compute summary edges for JAVA methods which return data values. Afterwards, a *two-pass slicing* algorithm [HBR90] for JAVA program is outlined.

The remainder of the paper has been arranged into the following five sections. The next section contains an overview on the creation of system-dependence graphs for object-oriented programs, while our approach for the representation of JAVA programs is described in Section 3. Section 4 presents algorithms for the computation of summary edges and for the static slicing of JAVA programs. In Section 5 we discuss the complexity of the algorithm, while Section 6 presents a summary and further work for the future.

2 Background

In [HBR90] Horwitz, Reps and Binkley have defined a graph (called system dependence graph) for precise interprocedural slicing. The system dependence graph (SDG) of a program contains the *program dependence graphs* of the procedures. The vertices of a program dependence graph (PDG) are the statements and predicate expressions of a given procedure, where vertices have been connected by two types of edges. The *data-flow* edges represent data dependences between the statements or expression, while the *control edges* show the dependences for the execution of statements represented by vertices.

The entry point of a procedure is represented by an *entry vertex* in the PDG. *Formal_in* and *formal_out* vertices are introduced in the PDG for denoting the parameter passing. A formal_in vertex is created for each formal parameter and global variable used in the procedure. There are formal_out vertices for those call-by-reference parameters and global variables which may be modified by the procedure. The call site in a procedure is represented by a *call vertex* and a set of *actual_in* and

actual_out vertices. The pairs of formal_in, actual_in and formal_out, actual_out vertices are connected by *parameter_in* and *parameter_out* edges respectively. In addition, there is a *calling edge* between a procedure call vertex and entry vertex of the called procedure's dependence graph.

Making use of the SDG graph interprocedural slices can be computed. In [HBR90] a method is described which computes precise slices in two-passes. This approach utilises the *summary edges* of the SDG graph. A summary edge represents the transitive flow of dependence occurring between an actual_in and actual_out vertex.

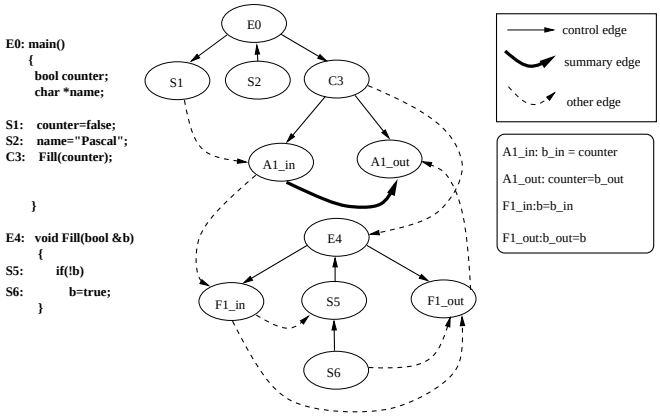


Figure 1.

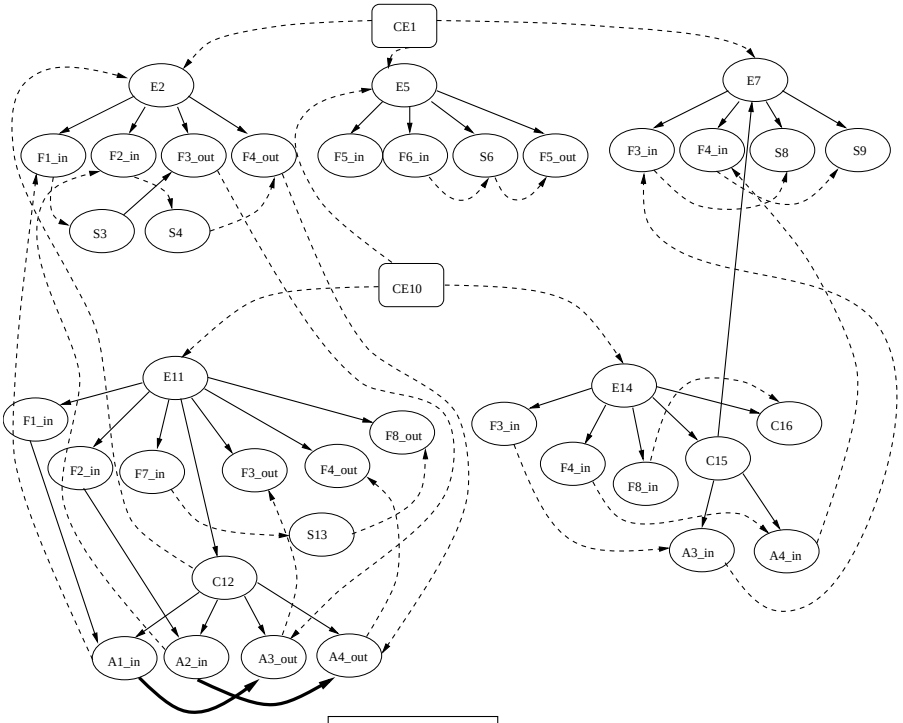
The return statements which return values to their callers are connected with parameter_out edges to their corresponding call vertices. In this case a summary edge is created from an actual_in vertex to the call vertex if the parameter corresponding to the actual_in vertex affects the returned value.

In Figure 1 a small C++ program and its SDG graph are depicted. The call-by-reference parameter of the function Fill may be modified during the execution of the function, hence there is a formal_out vertex for this parameter in the SDG graph. A summary edge is created from the vertex A1_in to the vertex A1_out, because the returned value of parameter b depends on the value of the actual parameter b.

In [LaH96] Larsen and Harrold introduced a method for the representation of C++ classes by SDGs. In their approach a *class-dependence graph* (ClDG) for each class is created. A class contains the *instance variables* for the implementation of the object's states and *methods* which execute operations on these states. The ClDG represents the data- and control dependences of a single class. In a ClDG the methods are implemented with PDGs, the only difference being that the entry vertices of these PDGs are called *method entry* vertices. A ClDG contains a class entry vertex which is connected to entry method vertices by *class member edges*. The parameters for the method are represented in the same way as that described above.

The instance variables can be considered as global variables, but the construction of vertices for these variables in constructor and destructor methods is different from other methods. There is no formal_in vertex for an instance variable in the class constructor or formal_out vertex for an instance variable in the class destructor.

The ClDG of a derived class contains a *class entry* vertex which is connected by *class member edges* to the method-entry vertices of this class. Furthermore, there is a class member edge from the class-entry vertex to a method entry vertex of the base class, if that method is inherited from its derived class.




```
CE1:   class Person
       {
         char *name;
         char *PID;
         Person nextperson;
       public:
E2:      Person(char *name, char *PID)
         {
S3:        this.name=name;
S4:        this.PID=PID;
         }
E5:      void AddNextPerson(Person person)
S6:      { nextperson = person; }
E7:      void PrintData()
S8:      {cout<<"Name: " <<name<<"\n";
S9:       cout<<"PID: "<<PID<<"\n";
         }
       }

CE10:  class SexPerson : public Person
       {
         bool sex;
       public:
E11:     SexPerson(char *name, char *PID,bool sex):Person(name,PID)
C12:     {
S13:       this.sex=sex;
         }
E14:     void PrintData()
C15:     {Person::PrintData();
S16:      cout<<"Sex :"<<sex<<"\n";
         }
       }
```

control edge
summary edge
other edge

F1_in	name=name_in	F7_in	sex=sex_in
F2_in	PID=PID_in	F7_out	sex_out=sex
F3_in	name=name_in	F8_in	sex=sex_in
F3_out	name_out=name	F8_out	sex_out=sex
F4_in	PID=PID_in	A1_in	name_in=name
F4_out	PID_out=PID	A2_in	PID_in=PID
F5_in	nextperson=nextperson_in	A3_out	name=name_out
F5_out	nextperson_out=nextperson	A4_out	PID=PID_out
F6_in	person=person_in	A3_in	name_in=name
		A4_in	PID_in=PID

Figure 2.

In Figure 2 two small C++ classes and their corresponding ClDGs are displayed. The vertices CE1 and CE10 denote the class-entry vertices of the classes Person and SexPerson, respectively. The vertices E2, E5, E7, E11, E14 are the method entry vertices. The method AddNextPerson from the class Person is inherited from the class Person. This fact is displayed with the $CE10 \rightarrow E5$ class member edge in the ClDG of SexPerson. The constructor of SexPerson calls the Person's constructor. The vertex C12 denotes the calling of this procedure. The parameters of the PrintData method are instance variables which can be represented in the ClDG as global variables. This method does not modify the value of the parameters, so there is no formal_out vertex for these parameters in the ClDG.

In [LaH96] Larsen and Harrold proposed a method for the representation of *polymorphic method calls* in ClDGs. In the case of a polymorphic method call, the type of referenced object cannot be determined at the compile time. The method uses a *polymorphic choice vertex* to represent the possible destinations of a polymorphic call.

From the call vertex of a polymorphic call there is an edge to the polymorphic choice vertex. From

the polymorphic choice vertex there is an edge to each call vertex. These call vertices are connected to the method entry vertices that represents a possible destination of the polymorphic call. While it is clear that this approach is imprecise in many cases, a precise solution to the type-inferencing problem is NP-hard [PaR94]. But using this approach the size of the SDG can increase significantly.

In Figure 3 the handling of polymorphic calls is illustrated. At compile time, the type of the object *psomeone* cannot be determined. Therefore, at the vertex C17, a polymorphic call is invoked for the method PrintData. The vertex P1 denotes the polymorphic choice and two empty vertices illustrate the call vertices.

Larsen and Harrold investigated the problem of the representation of incomplete systems (e.g. Class libraries). In their approach a universal driver (called a frame) was used to simulate the calling of available public methods which could be performed in any order.

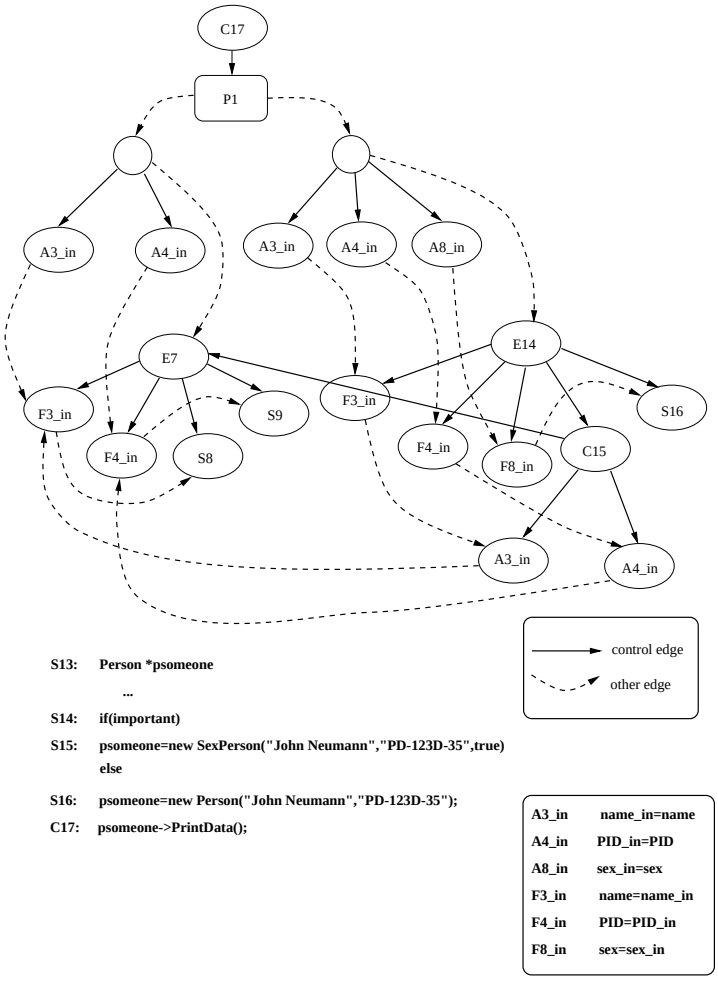


Figure 3.

3 Representation of JAVA programs

Every JAVA program consists of packages and every package consists of units. The units can include classes and interfaces, and a JAVA class is a collection of methods.

First of all, a simple case is dealt with where the JAVA programs consist of one package, and after

that our represention for JAVA programs is extended to those consisting of two or more packages. This is necessary because the fields of classes can affect their inheritance in different ways depending on whether an ancestor class is in the same package as the descendant's.

The JAVA programs and C++ programs are very similar so Larsen and Harrold's representation of C++ programs forms the basis of our work. Of course Larsen's representation cannot be wholly applied to JAVA programs because there are some differences between the two languages. In JAVA there are interfaces, and a class can include static variables and constructors of classes can be called implicitly. In C++ a class can have more than one ancestor in contrast to JAVA's class which can only have one. Section 3.1 describes our system-dependence graph for JAVA programs in general, while 3.2 goes over representations of interfaces. Section 3.3 gives a new representation for polymorphic calls, while 3.4 elaborates on the representation of static variables.

3.1 Representation in general

In the SDG which represents a JAVA program the inherited methods are not indicated by class member edges directly. A tree structure is obtained from which one can easily compute which methods belong to which class. In order to compute this it is sufficient to know the visibility of the methods in the case of one package. For JAVA programs consisting of two or more packages one must know where the classes were defined. Section 3.1.1 shows a representation of JAVA programs consisting of one package, while in 3.1.2 it is extended to cover multiple-package programs.

3.1.1 JAVA programs consisting of one package

A JAVA class (1) consists of static variables that implement the class's state, (2) instance variables that implement the objects' state, and (3) methods that implement the operations on the objects' and class's states. From this point on subclass and superclass denote the inherited class and ancestor class. The ClDG introduced by Larsen and Harrold is the basis of our work. So our representation also contains class-entry vertices and each method has a method-entry vertex. In Larsen's representation a class-entry vertex was connected to every method entry vertex by class member edges whose methods were elements of the class or were inherited. In C++ a complicated class hierarchy can be induced so the inheritance directly indicated by an edge was justified. In JAVA there is an Object class that is a superclass of every class of JAVA, because the compiler considers it an ancestor of the classes whose definitions do not contain an ancestor. If a class's definition contains an ancestor, Object is already a superclass of that ancestor. Hence it follows that the only base class in JAVA is the class Object, so we will not deal with the base and the derived classes's representation in separately.

In JAVA every method can have one of the following visibilities: private, protected, public or none of them (default)[1]. Every JAVA class hierarchy must be a tree structure, as every class can have at most one ancestor. It is known that in tree structures the ancestors of a vertex can be computed in linear time so the inheritance with class member edges will not be indicated directly, but each class member edge will be labelled with a private, protected or public label (sometimes it may be missing in which case it is the default case) according to the visibility of the methods. If class C2 is the immediate superclass of class C1 then the class-entry vertices of C1 and C2 are connected by a *class-dependence edge*. Notice that the class-entry vertices and class-dependence edges form a tree is due to the structure of JAVA class hierarchies. The methods belonging to a class C can be easily determined, because upwards in the tree all ancestors of C will be found and those methods are inherited, which are not connected to their defining class's entry vertex by a private class member edge (we suppose the program only consists of one package). A number of class member edges can omitted because of the new representation being used. The class Object is the root of every JAVA class hierarchy. Our

[1] For more detail see [JLS95]

representation does not contain the ClDG of this class except the Object class entry vertex which is connected to other class entry vertices.

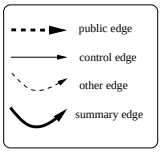
Let us construct a program-dependence graph for each method's body. The ClDG construction connects each method entry to the formal-in and formal_out vertices with control dependence edges according to the parameter list and global variables. In the JAVA programs the static and instance variables can be global variables. The representation of the C++ classes omits the formal_in vertices for instance variables in the class constructor, and the formal_out vertices for instance variables in the class destructor, because these variables cannot be referenced before they are allocated by the class constructor and afterwards they are deallocated by the class destructor. In JAVA programs static variables are allocated when the class is loaded and every instance of this class uses those static variables it has in common. Because the allocation of class variables precedes the allocation of the objects, the formal_in vertices for class variables are not omitted from the constructors. In JAVA the finalized methods receive the role of destructors. After reclaiming the storage occupied by their objects the class variables will be reallocated so the formal_out vertices for static variables are not omitted from the finalizers. Section 3.4 covers in detail the representation of the static variables.

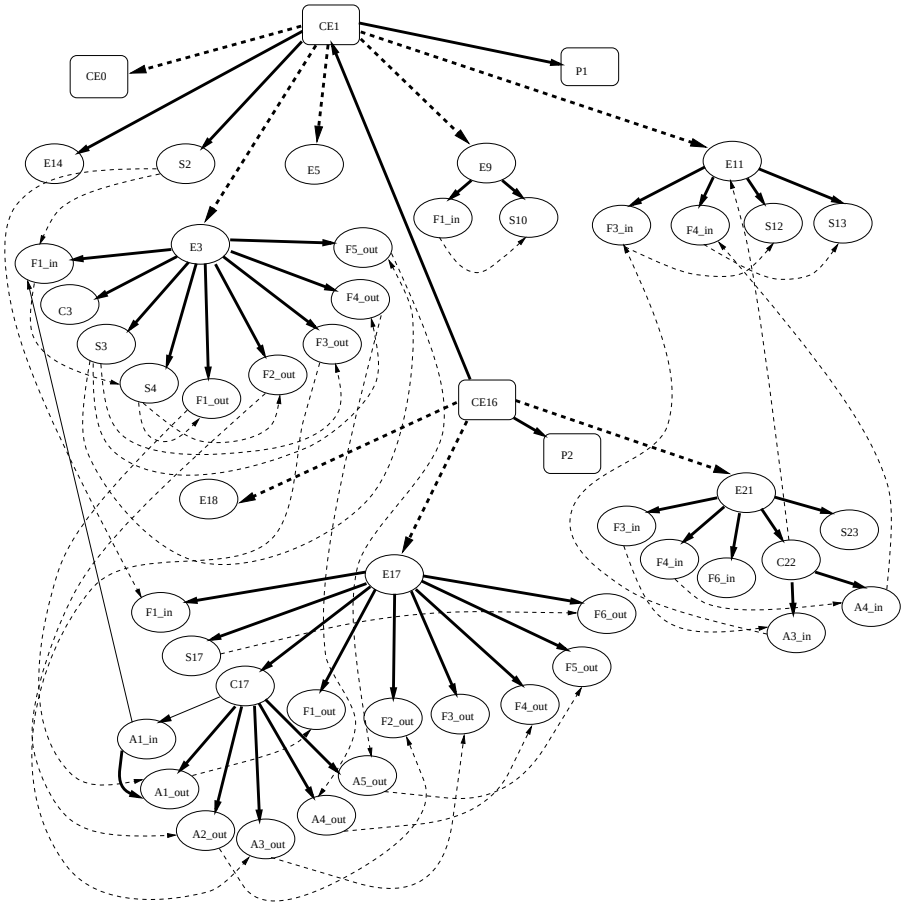
The first statement of a JAVA constructor may be an explicit call to another constructor of the same class, or an explicit call to a constructor of an immediate superclass. If an explicit constructor call is not present and the constructor being defined is not in the class Object, then the constructor body is implicitly assumed to begin with a call to a superclass constructor without arguments.

After a normal return of control from the superclass constructor all the instance variables that have initializers are initialized. The instance variables which do not have initializers all get default values. The PDG of a constructor is based on those rules outlined previously.

```
CE1:   class Person
       {
         private static int counter;
S2:      private static {counter=0};
         private int index;
         private String name;
         private Person nextperson;
E3:      public   Person();
S4:                { index=counter++;} ;
E5:      public  Person(String name, String PID)
C6:                { Person();
S7:                  this.name=name;
S8:                  this.PID=PID;
                   };
E9:      public static int GetCountNumber()
S10:               {return counter;}
E11:     public void PrintData()
                   {
S12:                 System.out.println("Name "+name);
S13:                 System.out.println("PID "+PID);
                   };
E14:     public  void AddNextPerson(Person person)
                   {
S15:                 nextperson=person;
                   };
       };
```

```
CE16:  class SexPerson extends Person
       {
         private String sex;
E17:     public SexPErson(){};
E18:     public SexPerson(String name,String PID, String sex)
                {
C19:              super(name,PID);
S20:              this.sex=sex;
                };
E21:     public  void PrintData()
                {
C22:              super.PrintData();
S23:              System.out.println("Sex "+sex");
                };
       };
```





F1_in	counter=counter_in	A1_in	counter_in=counter
F1_out	counter_out=counter	A1_out	counter=counter_out
F2_out	index_out=index	A2_out	index=index_out
F3_in	name=name_in	A3_out	name=name_out
F3_out	name_out=name	A4_out	PID=PID_out
F4_in	PID=PID_in	A5_out	nextperson=nextperson_out
F4_out	PID_out=PID	A8_in	lastperson_in=lastperson
F5_out	nextperson_out=nextperson		
F6_in	sex=sex_in		
F6_out	sex_out=sex		
F7_in	object=object_in		
F8_in	lastperson=lastperson_in		
F8_out	lastperson_out=lastperson		

Figure 4.

In Figure 4 CE0, CE1 and CE16 are the class entry vertices of the class Object, the class Person and the class SexPerson. $CE1 \rightarrow E3, CE1 \rightarrow E5, CE1 \rightarrow E9, CE \rightarrow E11, CE1 \rightarrow E14, CE16 \rightarrow E17, CE16 \rightarrow E18,$

$CE16 \rightarrow E21$ class member edges are addressed by the label public according to the visibility of these methods. Class SexPerson inherits method GetCountNumber() from class Person. This inheritance can be easily found from the ClDG of these classes because the $CE16 \rightarrow CE1$class dependence edge and $CE1 \rightarrow E9$class member edge form a path which represents this inheritance. The class SexPerson also inherits the method PrintData(). Since the superclass's name can be employed in a method access expression to access the superclass's methods, e.g. in our case Person.Printdata() denotes the inherited method in the class SexPerson.

In our representation the class Object does not have a ClDG. When an Object method is called (usually the Object constructor without parameter) the actual vertices must be omitted as the method's representation is missing. In the figure only the C3 vertex represents the implicit call of the constructor of the class Object. S3 vertex shows the initializers for the instance variables which don't get some value in the constructor so there are data-dependence edges between S3 and the formal-out vertices representing these variables, because the constructor does not change the value of the inicializers. The static variable *counter* belongs to both classes Person and SexPerson, while the F1_in formal_in vertex represents a variable counter in the PDG of the constructors. C17 represents the implicit call site of the constructor of the immediate superclass [2].

3.1.2 Representing JAVA programs consisting of more packages

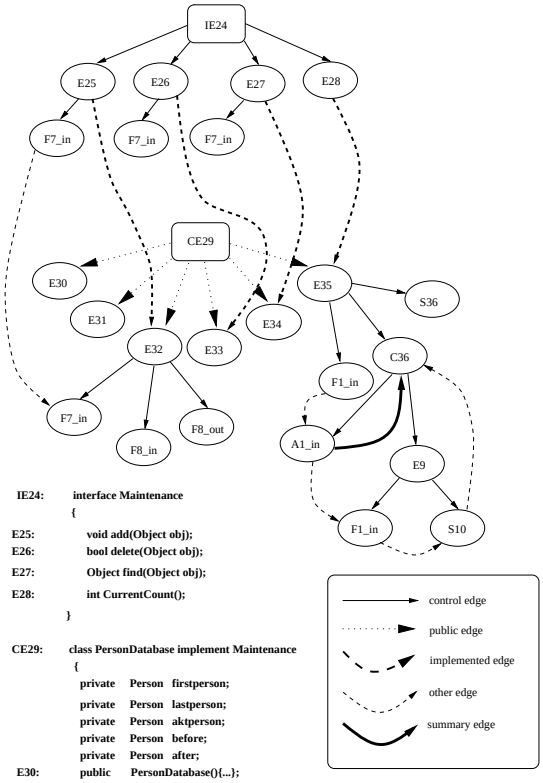
If the JAVA program consists of several packages it is necessary to know which package contains a given class, otherwise the inheritance cannot be precisely computed. In our representation we extend it to cover package entry vertices, where each *package entry vertex* represents a package. A package entry vertex is connected to class entry vertices belonging to this package by *package member* edges, that is if a package P contains a class C there is a package member edge between the P package and C class-entry vertices. The class C1 inherits the method M of class C2 if C2 is superclass of C1 and the class member edge between the M method-entry vertex and C2 class-entry vertex are addressed with a public or protected label, or the edge is not adressed but class C1 and class C2 belong to the same package. That is, the C1 and C2 class entry vertices are connected to the same package entry vertex.

In Figure 4, P1 and P2 package entry vertices depict two packages. P1 contains the class Person, while P2 contains the class SexPerson.

3.2 Representing interfaces

C++ does not contain interfaces, so we had to generalize the representation of Larsen and Harrold. A definition of an interface covers some set of methods but leaves their implementation unspecified. An interface can be implemented by one or more classes. If class C implements interface I this class must implement all methods of I. Here we construct an *interface dependence graph* (InDG) to represent the interfaces. An InDG contains an *interface-entry vertex* and method entry vertices denoting the methods of the interface. The interface-entry vertex is connected to the *method-entry vertex* for each method in the interface by an *interface member edge*. In the InDG formal_in vertices represent the parameters of each method, while formal_out vertices are omitted because the method's body is empty. If the M1 interface method is implemented by the M2 class method then the representation of M2 can contain more formal_in vertices than M1's because the body of M2 can use global variables. The M1 interface method-entry vertex is connected to the M2 class method-entry vertex by an *implemented edge* if M2 implements M1. Also, there are implemented edges among the formal_in parameters of methods M1 and M2. The representation of the interfaces is useful for computing an executable slice of a JAVA program.

[2] We notice that the representation of some methods are omitted from Figure 4




```
IE24:    interface Maintenance
         {
E25:        void add(Object obj);
E26:        bool delete(Object obj);
E27:        Object find(Object obj);
E28:        int CurrentCount();
         }

CE29:    class PersonDatabase implement Maintenance
          {
            private   Person  firstperson;
            private   Person  lastperson;
            private   Person  aktperson;
            private   Person  before;
            private   Person  after;
E30:        public    PersonDatabase(){...};
E31:        public    PersonDatabase(Person person){...};
E32         public     void add(Object obj)
                       {
                        lastperson.AddNextPErson(obj);
                        lastperson=obj;
                        lastperson.AddNextPerson(null);

                       };

E33:        public    bool delete(Obj obj){...};
E34:        public    Object find(Object obj){...}
E35:        public     int CurrentCount()
                      {
                          ...
C36,
S36:                  return Person.GetCountNumber();

                      };
         };
```

Figure 5.

Figure 5 shows our representation of the interface Maintenance. IE24 is the Maintenance interface entry vertex, CE29 the PersonDatabase class-entry vertex. E25 is the method entry vertex of the method Add(Object obj), and F7_in is a formal_in vertex for the parameter. E32 is the entry vertex of a class method which implements the previous interface method. The method E32 is a member of the class Person Database so the CE29 vertex is connected to the E32 vertex. The PersonDatabase implements interface Maintenance, this fact being represented by an implemented edge between the E25 interface method entry vertex and E32 class method entry vertex, and by an implemented edge between the formal_in vertices[3].

[3] We notice that the representation of some methods are omitted from Figure 5

3.3 Representing polymorphisms

Polymorphism permits, at runtime, a choice of one of a possible set of destinations of a method call. A static representation of a dynamic choice requires that all possible destinations be included. Larsen and Harrold use a polymorphic choice vertex to represent the dynamic choice among the possible destinations. A polymorphic choice vertex has call edges incident to subgraphs that represent calls to each possible destination. Because the number of destinations can be large these subgraphs can increase the size of the SDG drammatically.

In this work a polymorphic call will be represented as a normal call. If the possible destinations of a polymorphic call are methods F1 and F2 then they usually possess the same parameters. In spite of having the same parameter lists, the call representation of these methods can still be different because the bodies of the methods can include different global variables. Let Act1 be the set of actual_in and actual_out vertices which represent a call of the method F1, and let Act2 be a similar set for the method F2. Let Act be the union of Act1 and Act2. Sum1 stands for the set of summary edges which occur at the call sites of method F1, Sum2 the similar set for method F2. Let Sum be the union of Sum1 and Sum2. A polymorphic call of the methods F1 and F2 is represented by a call vertex C and the sets Act and Sum. The C vertex is connected to each actual vertex of the set Act by a control edge, and summary edges connect these vertices related to the set Sum. The C call vertex is connected to the F1 method entry and F2 method entry vertices via calling edges. An actual_in vertex from Act is connected to formal-in vertices of method F1 and method F2 similarly to a normal call. The most important difference between the representation of the polymorphic calls and normal calls is that in the first case two or more calling edges connect the call vertex to the possible destinations while only one edge is necessary in the second case. When the destinations of a polymorphic call consist of three or more methods the sets Act and Sum need to be defined in an analogous way.

Figure 6 shows a polymorphic call. At compile-time it is not known whether line C40 or line C41 will be executed, so either the E11 or E21 method will be called in line C42. If we represent a call of method E11 separated by the set of actual_in vertices, it will include A3_in and A4_in vertices, and the representation of the method E21 will include A3_in, A4_in and A6_in vertices. The union of these actual_in vertices consists of A3_in, A4_in and A6_in vertices. Because the first set is a proper subset of the second set the representation of this polymorphic call consists of the same vertices as the representation of method E21. Here the C42 call vertex is connected to two entry vertices (E11 and E21), in contrast with the representation of a normal call. Similarly A3_in and A4_in vertices are connected to formal_in vertices by two parameter_in edges.

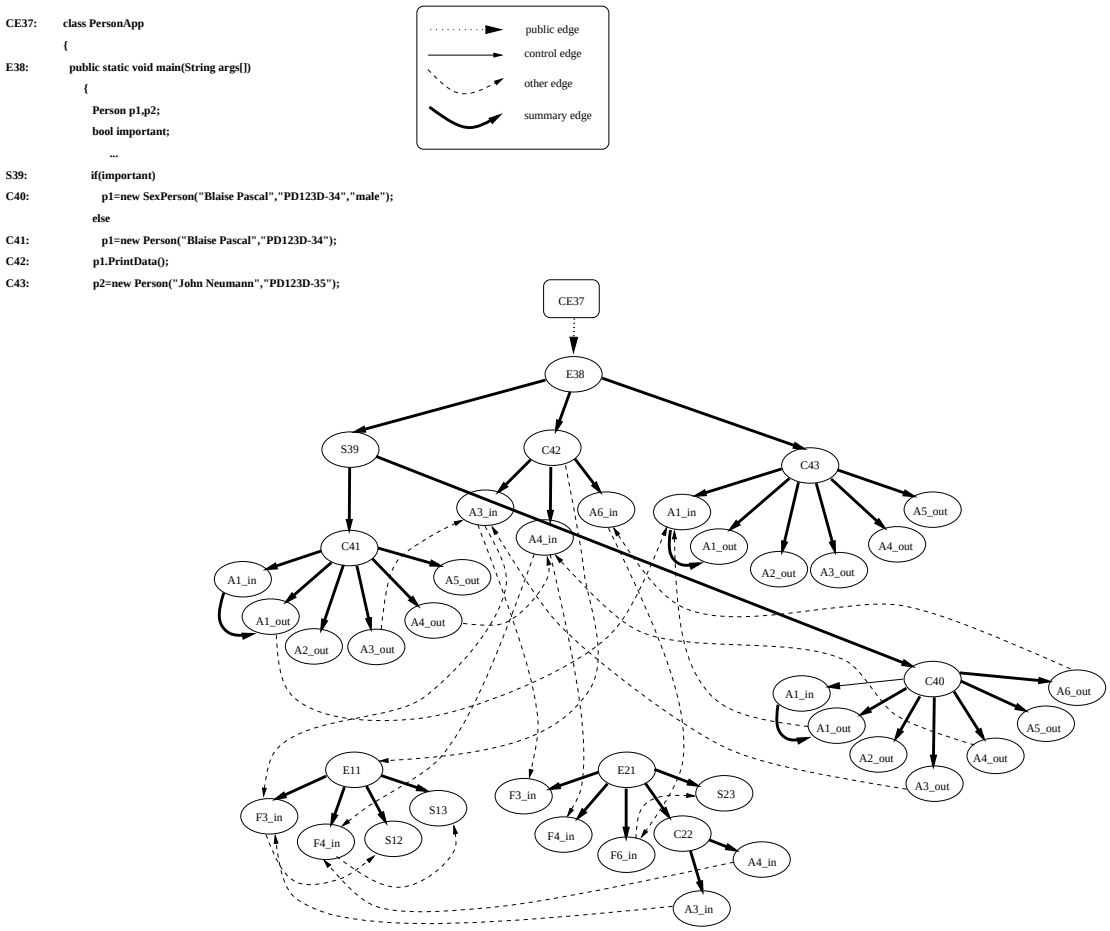


Figure 6.

3.4 Representing static variables

We mentioned in Section 3.1 that the representation of static variables differs from those of instance variables in constructors and destructors (in JAVA the *finalize* methods). A static variable is represented by a formal_in vertex in the constructors while an instance variable is not. As the initializers of the static variables are executed when the class is loaded, it is worth representing these initializers independently from the methods. The *assignment vertices* which represent the static initializers are connected to the class-entry vertex by *initialized edges*. There can also be data-dependence edges among these assignment vertices. Each formal_in vertex representing a static variable V is connected to the vertex which represents the initializer of variable V by a data-dependence edge, because at the creation of the first object the constructor receives the values of static variables whose values are computed by the initializers. In Figure 4 the vertex S2 depicts the initializer of the variable counter. This vertex is connected to the CE1 class entry vertex by an initialized edge. In the representation of constructors the F1_in vertex denotes the variable counter, so the S2 and F1_in vertices are connected by a data-dependence edge.

In the system-dependence graphs the changes of states of the objects are represented by data-dependence edges between the actual_in and actual_out vertices. For instance variables these actual vertices must belong to the same object, because whenever a new instance of class (a new object) is allocated, a new variable is created for every instance variable of that class. For static variables there

is exactly one variable of that name, no matter how many instances of the class are created, so an actual_in vertex can be connected to an actual_out vertex despite the fact that the objects including these vertices are two different instances of a class. Naturally, the actual_in and actual_out vertices must represent the same variable in both cases.

In Figure 6 the program contains two different objects (p1 and p2), the constructor of object p1 (C40) being connected to the constructor belonging to object p2 (C43) by a data-dependence edge (A1_out→A1_in), because the variable counter has exactly one instance and objects p1 and p2 have this instance in common.

4 Slicing JAVA programs

We described in the previous section how JAVA programs can be represented by an SDG. Horwitz, Reps and Binkley [HBR90] compute interprocedural slices by solving a graph reachability problem on an SDG, their algorithm uses the summary edges. Reps et al give an efficient algorithm for calculating these summary edges [RHS94], however, this algorithm can only be applied to those programs which only include procedures. Void methods do not return a value, so these methods can be represented as procedures. The representation of a method returning some value differs from procedure representations. The return vertices are connected to the call vertices by parameter_out edges and there can be a summary edge between an actual_in and call vertex, that is, the call vertex can be considered as the actual_out vertex for the return value. Section 4.1 gives a modified algorithm which computes summary edges for programs containing functions (methods). This algorithm can be applied to JAVA programs as well. Section 4.2 describes how JAVA programs can be sliced up.

4.1 The modified algorithm

Ball and Horwitz [BaH93] solve the problem of slicing programs with arbitrary control flow. They developed a representation of the break statement in the PDG according to the control after this statement. This representation can be applied to return statements too. However, there is an important difference between the break and return representations. A return statement can contain variables (return value of the method) so it can have a data-dependence on other statements, that is, the set of the ingoing edges of a return vertex may consist of data and control-dependence edges. The set of outgoing edges of a return vertex consist of only control-dependence edges.

```
[1]  void Propagate(e:edge)
[2]      {if(e not member of PathEdge)
[3]          {insert e into PathEdge; insert e into WorkList;}
         }
[4]  (set of edges) ComputeSummaryEdges(G:SDG)
[5]      { declare PathEdge, SummaryEdge, WorkList :  set of edges;
[6]        PathEdge=0; SummaryEdge=0; WorkList=0;
[7]        for(w first member of FormalOutandReturnVertices,w!=0,
               w next member of FormalOutandReturnVertices)
[8]         {insert(w→ w) into PathEdge; insert(w→ w) into WorkList; }
[9]         while(WorkList!=0)
[10]             {select and remove an edge v→w from WorkList;
[11]               switch(type of v)
[12]                      {case return :
[13]                         {if(v==w)
[14]                           {for(x first vertice that type of x→v is flow
```

```
                                    or control, x!=0, x next such vertices)
[15]                                 Propagate(x→ w);}
[16]                              else
[17]                               {for(x first vertice that type of x→v is control,
                                    x!=0, x next such vertices)
[18]                                 Propagate(x→ w);}
[19]                              break;}
[20]                            case actualout:
[21]                              {for(x first vertice that type of x→v is control
                                   or summary x!=0, x next such vertices)
[22]                                 Propagate(x→ w);} break;}
[23]                            case formal-in:
[24]                              {for(c first member of Callers(Proc(w)), c!=0,
                                    c next member)
[25]                                 {x=CorrespondingActualin(c,v);
                                     y=CorrespondingActualOutorCall(c,w);
[26]                                  insert x→y summary edge;
[27]                                  for(a first vertice that y→a member PathEdge,
                                        next such member of PathEdge)
                                      Propagate(x→a);break;}
[28]                            default:
[29]                               {for(x first vertice that type of x→v is flow
                                    or control, x!=0, x next such vertice)
                                    Propagate(x→ w);}
              } return SummaryEdge;
      }
```

Figure 7.

Horwitz et al.'s speeding-up algorithm starts from formal_out vertices and computes the transitive closure of the data and control-dependence edges. When the algorithm computes a summary edge new path edges can be induced (23..27) where the path edges are introduced in order to represent the computed transitive closure.The new algorithm starts from the formal_out and the return vertices. If an return vertex R2 is reached during the computation there must be a vertex S which is control dependent on the vertex R2 (there is control edge between R2 and S) and being reached by the algorithm. After the execution of statement R2 the function returns the contol to the caller independently of the variable-values occurring in R2, so S does not have data dependence on the vertices which are connected to R2 via data dependence edges. For this reason the modified algorithm ignores the data-dependence edges going into R2. Of course, when the algorithm starts from the return statements all edges going into these vertices are considered.

Figure 7 shows the modified algorithm. We extend the basic algorithm with lines 12-19. The algorithm uses the following additional functions: *Proc* returns that function which contains the given vertex, *Callers* returns the set of vertices which call the given vertex.

4.2 Slicing JAVA program

A slicing can have two different goals: (1) The slice consists of the program components which may affect the value of a variable V at a program point p, or (2) The slice is an executable program which gives the same value sequence at a program point p for variable V as the original program for any input. The slices are computed with respect to a slicing criterion which consist of a program point p

and variable V that is defined or used at p. Larsen and Harrold [LaH96] use a slightly modified slicing criterion because object-oriented systems substitute multi-variable references with method calls that simply return a value. The modified slicing criteria $< p, x >$ consists of a statement p and variable or method call x. If x is a variable, it must be defined or used at p. If x is a method call it must be called at p.

We discussed in Section 3 how a JAVA program can be represented by an SDG. Horwitz et al. developed a two-pass algorithm which computes precise interprocedural slices with the aim of an SDG. Because the SDG representing JAVA programs includes more vertices and edge types than the original, their algorithm must be extended according to the changes. Both passes compute backward closures. The first pass starts from p and considers the data dependence, control dependence, summary, parameter_in, calling, class member, interface member and implemented edges. If the algorithm reaches a vertex, then the vertex will be indicated. The second pass starts from the signed vertices and considers the data dependence, control dependence, summary, parameter_out edges.

```
[1]   void ComputeExecutableSlice(G:SDG)
[2]        { for(l first InDG from SDG, l!=0, l next InDG from SDG)
[3]            { l_entry=interface entry of l;
[4]                M_entries=interface method entries which members of l;
[5]                for(M first method entry from M_entries, M!=0,
                      next method entry from M_entries)
[6]                   {ParamOfM=formal-in vertices which are
                       connected M by control edges;
[7]                    Implement=ClDG-s from SDG which implement l;
[8]                    for(C first ClDG of Implement, C!=0, next ClDG from Implement)
[9]                       { if(exist e implement edge from M to C)
[10]                           {CM=target of e implement edge;
                                   /* CM method entry vertice */
[11]                            ParamofCM=formal-in vertices which are connected
                               CM by control edges;
[12]                            if(ParamofCM proper subset of ParamofM)
[13]                              {Create copy of ParamofM - ParamofCM in C;
[14]                                 Connect these vertices to CM by control edges;
                                 };
                               }
[15]                         else
[16]                            {Create CM method entry vertice;
[17]                              Connect CM to M by implement edge;
[18]                              Create copy of ParamofM into C;
[19]                              Connect these vertices to CM by control edges;
           }}}}}
```

Figure 8.

The vertices reached by the modified algorithm are members of the interprocedural slice. This slice satisfies the goal (1), but usually is not executable [HoR92]. Horwitz and Reps give an algorithm to extend a slice satisfying (1) to an executable slice satisfying (2). However, in case of Java programs after this transformation the extended slice can not be executed.

In Section 3.2 we mentioned that the representation of interfaces plays an important role in slicing executable programs. Let S1 be the set of vertices (statements and predicates) belonging to the slice

obtained by the Horwitz and Reps algorithm. It is possible that the class C1 and class C2 implement an interface which includes method F1 (type1 a, type2 b) and method F2(), where type1 and type2 can be any type. Method F1 and F2 are implemented by method E11 and E12 in class C1 and by method E21 and E22 in class C2 (E11, E12, E21, E22 denote the method entry vertices of the methods from C1 and C2). If slice S1 contains the E11 and E22 method entries but does not contain E12 and E21 then the slice is not executable, because F1 and F2 belong to the slice S1, class C1, and class C2 implement interface I in the future too but in the slice both C1 and C2 implement only one method from I that will cause a compile-time error. A similar problem can be induced if different class methods implementing the same interface method have a different parameter list after the slicing. Our algorithm in Figure 8. solves all the above problems. The input of the algorithm is a system-dependence graph of a slice which is transformed by the Horwitz and Reps algorithm.

The algorithm examines the implementations of interfaces belonging to the slice. If method M1 is a member of interface I and this method is not included in a class C which implements I, then the algorithm adds method M2 to C whose method will implement M1. The body of M2 will be empty (lines 16-19). If C contains a method M2 which implements method M1 and the parameter list of M2 consists of less parameter list of M1 the algorithm completes than the parameter list of M2 related to the parameter list of M1(lines 13-14).

5 The size of the SDG and the cost of the algorithm computing an executable slice

Larsen and Harold [LaH96] give an estimate of the size of the SDGs representing object-oriented programs. We can apply their approximation here because our represention is similar to theirs. Figure 9 lists the variables that contribute to the size of the SDG.

```
Vertices        Largest number of statements in a single method
Edges           Largest number of edges in a single method
Params          Largest number of formal parameters for any method
ClassVar        Largest number of class variables in a class
ObjectVar       Largest number of instance variables in a class
ClassSites      Largest number of call sites in any method
TreeDepth       Depth of the inheritance tree
Methods         Number of methods of the system
```

Figure 9.

We give a bound on the number of parameters for any method *(ParamVertices)*, the size of the methods (*Size*) and the size of the SDG (*Size(SDG)*).

```
(1) ParamVertices=Params+ObjectVar+ClassVar
(2) Size=O(Vertices+CallSites*(1+TreeDepth*(2*ParamVertices))+2*ParamVertices)
(3) Size(SDG)=O(Size*Methods)
```

Figure 10.

Figure 10 shows that *size(SDG)* provides a rough upper bound on the number of vertices in an SDG. In practice an SDG may be considerably more space efficient.

We believe that our modifications give more efficient representation than the original one. Firstly, in the representation of Larson and Harrold [LaH96] all inheritance is indicated by distinct edges. Because there can be a lot of inheritance in complicated JAVA programs the number of edges can increase substantially, so the variable **Edges** (Figure 9) can be different in the two representations. The reader should note that our representation uses only class-dependence and package-member edges to compute the inheritances, and the number of these edges can be significantly less than class member edges. Our representation also uses fewes edges and vertices to represent a polymorphic call, hence the value of the variables **Edges** and **Vertices** have been decreased.

The cost of the ComputeExecutableSlice can be bounded by the variables in Figure 11.

```
Interfaces              Number of the interfaces in the system
MethodsofInterface      Largest number of methods in any interfaces
Implement               Largest number of the classes implementing any interfaces
```

Figure 11.

The total cost of our algorithm is bounded by

```
O(Interfaces*MethodsofInterface*Implement*Params)
```

The complexity was computed on the basis of lines 2, 5, 8, 13 and 18 in Figure 8.

6 Summary

We have now presented a method for the interprocedural static slicing of JAVA programs. Even though our approach is based on the representation method of object-oriented programs developed in [LaH96], by using special JAVA features the efficiency of the representation can be somewhat improved. Besides this the implementation of the hierarchy of JAVA classes was made simpler. A technique was proposed for the correct representation of static variables of JAVA classes in SDGs, our approach being able to produce precise slices for JAVA programs containing several packages and interfaces. We then introduced a new representation in SDGs for polymorphic calls, an approach reduces the number of extra vertices required for handling polymorphism. Afterwards we presented an algorithm for the computation of summary edges at call sites, the summary edges being utilised in the two-pass algorithm for static slicing of JAVA programs. We have already developed a system for the static slicing of C programs, and are currently modifying this system using the method presented in this paper.

References

[BaH93] Ball T., Horwitz S.: *Slicing Programs with Arbitrary Control Flow.* Proceedings of the First International Workshop on Automated and Algorithmic Debugging, P. Fritzson, Ed., vol. 749 of Lecture Notes in Computer Science, Springer-Verlag, pages 206-222.

[BaH93] Bates S., Horwitz S.: *Incremental Program Testing Using Program Dependence Graphs.* Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South California, 1993. ACM Press,pages 384.-396,1993.

[Bin91] Binkley D.:*Multi-procedure program integration.* Ph.D. Thesis and Tecnical Report 1038, Department of Computer Science,Univertsity of Wisconsin, Madison, 1991.

[FGK91] Fritzson P., Gyimóthy T., Kamkar M., and Shahmehri N.: *Generalized Algorithmic Debugging and Testing.* Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 26(6): pages 317-326, 1991.

[FoG96] Forgács I., Gyimóthy T.: *A Fast Interprocedural Slicing Method for Large Programs.* submitted to POPL'97.

[GaL91] Gallagher K. B., Lyle J. R.: *Using Program Slicing in Software Maintance.* IEEE Transactions on Software Engineering, 17(8), pages 319-349, 1987.

[JLS95] *The JAVA Language Specification*, Sun Microsystems Computer Corporation, 1995.

[JZR91] Jiang J., Zhou X., Robson D.:*Program Slicing for C - The Problems In Implementation.* Proceedings of the Conference on Software Maintanance (1991), pages 182-190.

[HBR90] Horwitz S., Reps T., Binkley D.: *Interprocedural Slicing Using Dependence Graphs.* ACM Transactions on Programming Languages and Systems, 12, 1, pages 26-61, 1990.

[HoR92] Horwitz S., Reps T.: *The Use of Program Dependence Grapgh in Software Engineering.* Proceedings of the 14th International Conference on Software Engineering, 1992.

[HPR89] Horwitz S., Pfeffer P., Reps T.: *Dependence Analysis for Pointer Variables.* Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation, pages 28-40, 1989.

[HoP89] Horwitz S., Prins J., Reps T.: *Integrating non-interfacing versions of programs.* ACM Transactions on Programming Languages and Systems, 11(3), pages 345-387, 1989.

[Kam93] Kamkar M.: *Interprocedural Dynamic Slicing with Applications to Debugging and Testing.* Linköping Studies in Science and Technology - Dissertations No. 297,Department of Computer and Information Science, Linkping Universitz, 1993.

[LaH96] Larsen L., Harrold M. J.: *Slicing Object-Oriented Software.* Proceedings of 18th ICSE Conference, pages 495-505, 1996.

[LiC92] Livadas P. E., Croll S.: *Static program slicing.* Tecnical Report SERC-55f, University of Florida, Software Engineering Research Center, Computer and Information Sciences Department, 1992.

[LMR92] Lejter M., Meyers S., Reiss S. R.: *Support for Maintaining Object-oriented Programs.* IEEE Transactions on Software Engineering, 1992.

[MMK94] Malloy B. A., McGregor J. D., Krishnaswamy A., Medionda M.: *An Extensible Program Representation for Object-oriented Software.* ACM Sigplan Notices, 29(12), pages 38-47, 1994.

[OtO84] Ottenstein K. J., Ottenstein L. M.: *The Program Dependence Graph in a Software Development Environment.* Proccedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practial Software Development Environments, pages 177-184, 1984.

[PaR94] Pande H. D., Ryder B. G.: *Static Type determination for C++.* Technical Report LCSR-TR-197, Rutgers Univertsity, 1993.

[PGH94] Paakki J., Gyimóthy T., Horváth T.: *Effective Algoritmic Debugging for Inductive Logic Programming.* 4th International Workshop on Inductive Logic Programming (ILP-94) (S. Wrobel, ed.) Bad Honnef/Bonn, 1994. GMD-Studien Nr. 237, Gesellschaft fr Mathematik und Datenverarbeitung, pages 175-194, 1994.

[RHS94] Reps T., Horwitz S., Sagiv M., Rosay G.: *Speeding up Slicing.* Proceedings of Second ACM Conference on Foundations of Software Engineering, pages 11-20, 1994.

[RoH94] Rothermel G., Harrold M. J.: *Selecting Regression Tests for Object-orinted Software.* Proceedings of Conferece on Software Maintance, pages 14-25, 1994.

[Wei84] Weiser M.: *Program Slicing.* IEEE Transactions on Software Engineering, 10, 4, pages 352-357, 1984.