

Slicing Objects Using System Dependence Graph

Donglin Liang and Mary Jean Harrold

Department of Computer and Information Science

The Ohio State University

E-mail: {dliang,harrold}@cis.ohio-state.edu

Abstract

We present an SDG for object-oriented software that is more precise than previous representations and is more efficient to construct than previous approaches. The new SDG distinguishes data members for different objects, provides a way to represent object parameters, represents the effects of polymorphism on parameters and parameter bindings, represents incomplete classes efficiently, and provides a way to represent class libraries. Based on this system dependence graph, we introduce the concept of object slicing and an algorithm to implement this concept. Object slicing enables the user to inspect the statements in the slice, object-by-object, and is helpful for debugging and impact analysis.

Keywords: *Slicing, system dependence graph, program analysis, object-oriented.*

1 Introduction

Program slicing has many applications in software maintenance, such as debugging, regression testing, program understanding, and reverse engineering (e.g., [5, 12]). A *program slice* [7], is a set of program statements and predicates that might affect the value of a program variable v that is defined or used at a program point p ; $\langle v, p \rangle$ is known as the *slicing criterion*. Horwitz et al. developed the *system dependence graph* (SDG) and a two-phase graph-reachability algorithm on the SDG to compute interprocedural slices [7]. Researchers have extended the SDG to represent various language features and proposed variations of dependence graphs that facilitate finer-grained slices (e.g., [1, 8]).

Object-oriented concepts, such as classes, objects, inheritance, polymorphism, and dynamic binding, however, make representation and analysis techniques, developed for imperative language programs, inadequate for use with object-oriented programs. Researchers have extended existing techniques to handle some of these features. Larsen

and Harrold [10] extend the SDG to represent some of the features of object-oriented programs. They introduce the *class dependence graph*, which can represent a class hierarchy, data members, and polymorphism. After the SDG is constructed using class dependence graphs, Horwitz et al.'s two-phase slicing algorithm can be used, with minor modification, for computing slices on object-oriented programs. One limitation of this approach is that it cannot distinguish data members for different objects instantiated from the same class; thus, the resulting slices may be unnecessarily imprecise. Tonella et al. [13] use an approach similar to Larsen and Harrold's except that, by passing all data members of an object as actual parameters when the object invokes a method, the approach can distinguish data members for different objects instantiated from the same class.

Although these existing approaches provide techniques for representing some features of object-oriented programs, there are several areas in which they can be improved. First, to facilitate the computation of more precise slices, the representation itself can be made more precise. For example, existing techniques for representing objects as parameters do not distinguish among data members in a parameter object; the existing technique for representing objects of several possible types may cause spurious dependences among statements that send messages to the objects. Second, to facilitate the application of slicing to larger programs, the representation can be constructed more efficiently. For example, because of the way in which data members are represented, existing approaches perform unnecessary data-flow computation. Finally, to correctly represent inherited methods that call virtual methods redefined in the derived class, the simple strategy to reuse representations of the inherited methods must be extended.

This paper presents an SDG for object-oriented software that is an extension of existing representations [10, 13]. This SDG more fully represents the features of object-oriented programs, including objects, that are present in languages such as C++ and Java. The paper also intro-

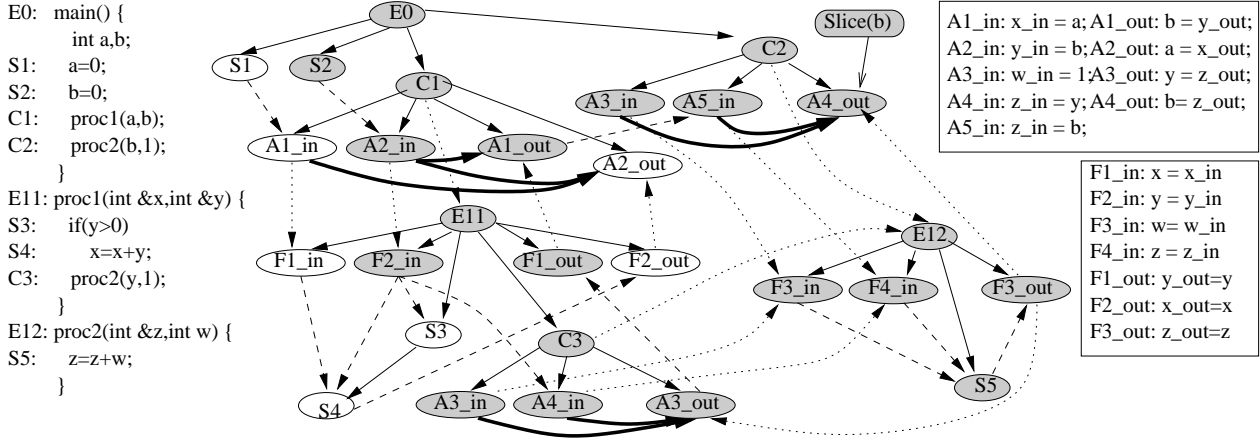


Figure 1. Program 1 and its system dependence graph.

duces the concept of *object slicing*, which identifies statements in methods of an object that might affect the slicing criterion — as if the object had its own copy of the methods. The main benefits of this SDG are that it distinguishes data members for different objects, represents objects that are used as parameters or data members in other objects, and represents the effects of polymorphism on parameters and parameter binding.¹ Another benefit is that our SDG-construction algorithm is more efficient than previous approaches, which may enable more efficient slicing on larger programs. A final benefit is that our technique for object slicing lets a user inspect statements in a slice, object by object.

2 Slicing Using SDGs

A *system dependence graph* (SDG) [7] contains one procedure dependence graph for each procedure. A *procedure dependence graph* [4] represents a procedure as a graph in which vertices are statements or predicate expressions. *Data dependence* edges represent flow of data between statements or expressions; *control dependence* edges represent control conditions on which the execution of a statement or expression depends. Each procedure dependence graph contains an *entry* vertex that represents entry into the procedure. To model parameter passing, an SDG associates each procedure entry vertex with formal-parameter vertices: a *formal-in* vertex for each formal parameter of the procedure; a *formal-out* vertex for each formal parameter that may be modified [9] by the procedure. An SDG associates each callsite in a procedure with a *call* vertex and a set of actual-parameter vertices: an *actual-in* vertex for each actual parameter at the callsite; an *actual-out* vertex for each

actual parameter that may be modified by the called procedure. At procedure entries and callsites, global variables are treated as parameters. These parameter vertices represent the assignments featuring the copy-in/copy-out parameter passing scheme; with these vertices, a data-flow analysis algorithm [2] can be used to compute data dependences among the parameters and statements in the procedures.

An SDG connects procedure dependence graphs at callsites. A *call* edge connects a call vertex to the entry vertex of the called procedure’s dependence graph. Parameter edges represent parameter passing: *parameter-in* (*parameter-out*) edges connect actual-in; formal-in vertices (formal-out and actual-out vertices).

Figure 1 shows Program 1 and its SDG. In the figure, ellipses represent program statements and parameter vertices. We refer to a particular parameter vertex by prefixing the parameter label with the call or entry vertex upon which it is control dependent. For example, C1→A1_in refers to the parameter vertex representing actual parameter *a* in the call to *proc1()* at C1. In the figure, solid lines represent control dependences, dashed lines represent data dependences, and dotted lines represent procedure calls and parameter bindings. For example, statement S4 is control dependent on the value of the predicate in S3; thus, there is a control dependence edge (S3,S4) in the SDG. The value of *b* in S2 is passed into *proc1()* at C1; thus, there is a data dependence edge (S2,C1→A2_in) in the SDG. A parameter binding occurs between *a* in *main* and *x* in *proc1()* at the call to *proc1()* at C1; this binding results in parameter-in edge (C1→A1_in,E11→F1_in) and parameter-out edge (E11→F2_out, C1→A2_out).

Horwitz et al [7] compute interprocedural slices by solving a graph-reachability problem on an SDG. To facilitate the computation of interprocedural slices only along paths corresponding to legal call/return sequences, an SDG uses summary edges to explicitly represent the transitive flow of dependence across call sites caused by data dependences,

¹Our representation also efficiently represents classes derived from, and objects instantiated from, library-defined classes, although, because of space limitations, we do not discuss these features here.

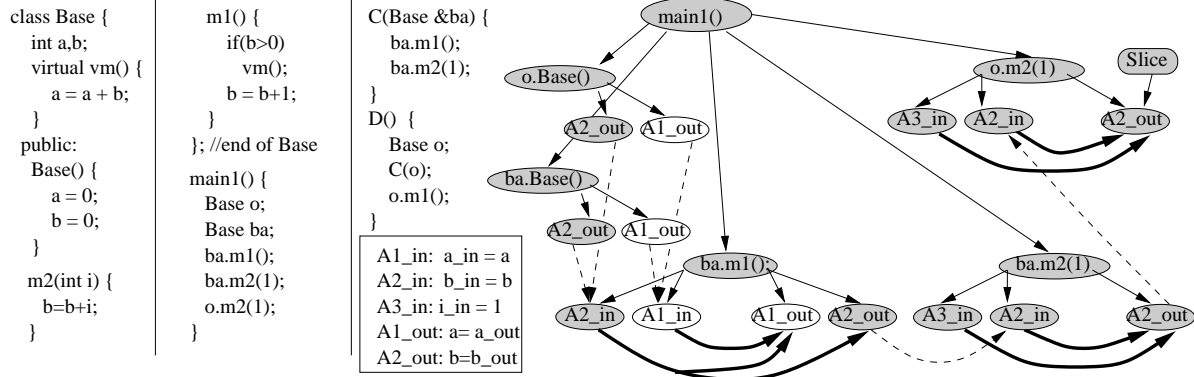


Figure 2. Program 2 on the left and its partial class dependence graph on the right.

control dependences, or both. A *summary edge* connects an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex may affect the value associated with the actual-out vertex. Thus, edge $(C1 \rightarrow A2_{in}, C1 \rightarrow A2_{out})$ represents the fact that, in procedure $proc1()$, the value of b that is passed to $proc1()$ affects the value of a that is returned by $proc1()$.

The slicing algorithm consists of two passes. During the first pass, the algorithm traverses backward along all edges except parameter-out edges, and marks reached vertices. During the second pass, the algorithm traverses backward from all vertices marked during the first pass along all edges except call and parameter-in edges, and marks reached vertices. The slice is the union of the marked vertices.

To illustrate, consider the computation of a slice for parameter b at callsite $C2$ (i.e., $C2 \rightarrow A4_{out}$). During the first pass, the algorithm marks vertices $C2 \rightarrow A4_{out}$, $C2 \rightarrow A3_{in}$, $C2 \rightarrow A5_{in}$, $C2$, $C1 \rightarrow A1_{out}$, $C1 \rightarrow A2_{in}$, $C1$, $S2$, and $E0$. During the second pass, the traversal starts from the vertices marked in pass one, and the algorithm marks vertices $E12$, $E12 \rightarrow F3_{out}$, $S5$, $E12 \rightarrow F3_{in}$, $E12 \rightarrow F4_{in}$, $E11$, $E11 \rightarrow F1_{out}$, $C3 \rightarrow A3_{out}$, $C3$, $C3 \rightarrow A3_{in}$, $C3 \rightarrow A4_{in}$, $E11 \rightarrow F2_{in}$. Figure 1 depicts the resulting slice using shaded vertices.

3 Limitations of Existing Techniques

An object contains a set of data members and methods. An object can receive messages and invoke its methods to handle those messages. An invoked method requires access to the data members of the object. Larsen and Harrold [10] use additional parameters to represent the data members referenced by a method, when the method is invoked to handle a message. In this way, the data dependence between two consecutive method calls, introduced by data members, can be represented as the data dependence between the actual parameters at the method callsites. Figure 2 shows Program 2 and the portion of the SDG for func-

tion $main1()$ using this approach. In the figure, for example, there are data dependence edges between the actual-out parameter vertices representing data members a and b (i.e., $ba.Base() \rightarrow A1_{out}$, $ba.Base() \rightarrow A2_{out}$) for method call $ba.Base()$ and the actual-in parameter representing a and b (i.e., $ba.m1() \rightarrow A1_{in}$, $ba.m1() \rightarrow A2_{in}$) for method call $ba.m1()$. This representation lets the slicer omit, from the slice, data members that cannot affect the value of the variable(s) being sliced, thus, gaining precision. For example, in the figure, the shaded vertices belong to the slice that begins at $o.m2(1)$'s callsite for data member b (indicated in the figure by $o.m2() \rightarrow A2_{out}$); none of the vertices that define data member a , which cannot affect the computation of b 's value, are in the slice.

One limitation of this approach is that the data dependences obtained using the approach for creating the individual procedure dependence graphs are imprecise: by treating data members declared in a class as if they were global to the methods of that class, the approach fails to consider the fact that in different method invocations, the data members used by the methods might belong to different objects.² For example, in $main1()$'s representation in Figure 2, there is an edge from $o.Base() \rightarrow A1_{out}$ (representing $o.a$) to $ba.m1() \rightarrow A1_{in}$ (representing $ba.a$), which is a spurious data dependences. A second limitation of the approach is that it does not handle cases in which an object is used as a parameter or as a data member of another object. Thus, $C(Base \&ba)$ of Program 2 cannot be represented.

Tonella et al. [13] address the first limitation by extending a method's signature to include data members of the class as formal parameters so that an object can pass its data members into the method as actual parameters. Their approach, however, is unnecessarily expensive because each method callsite has actual parameter vertices for all data members of the object, even if only a few of them are ref-

²Grove et al. address this problem for call-graph construction using class contours [6]. This approach, however, has not been used to construct more precise representations for SDGs.

erenced by the method. They address the second limitation by representing an object as a single vertex when the object is used as a parameter. This representation, however, might cause the slicer to produce imprecise slices. For example, in Program 2, if the slicing criterion is $o.b$ at the end of $D()$, then o in $C(o)$ must be included in the slice. This situation requires the slice to include ba in function $C(Base \&ba)$, which in turn requires the slice to include both $ba.a$ and $ba.b$. However, from the program, we can see that $ba.a$ does not affect $o.b$.

Both of the above approaches [10, 13] perform unnecessary computation while calculating data dependences. The class dependence graph construction algorithm uses the data dependence graph construction algorithm that is used in the construction of a traditional SDG: if a statement S in method $A :: m_1$ calls method $B :: m_2$, and $B :: m_2$ modifies B 's data member b , then the algorithm propagates a definition of b , which is introduced at the actual-out parameter representing b at the callsite, throughout m_1 . However, no statement in m_1 , except call statements to B 's methods, will use b . Similar problem exists in Tonella et al.'s approach, in which definitions of data members of objects are propagated throughout the program.

Polymorphism, another important object-oriented concept, lets the type of an object be decided at runtime; we refer to such an object as a *polymorphic object*. This feature is difficult to handle during static analysis: when a polymorphic object receives a message, the selection of a message-handling method depends on the runtime type of the object. In some existing techniques [10], a message sent to a polymorphic object is represented as a set of callsites, one for each candidate message-handling method, connected to a *polymorphic choice* vertex with *polymorphic choice* edges. This approach may give incorrect results: in function $main2()$ (shown in Figure 3), the approach uses only one callsite to represent statement “ $(*p).m1()$ ” because $m1()$ is declared only in $Base$. However, when $m1()$ is called from objects of class $Derived$, it invokes $Derived :: vm()$ to modify d , and when $m1()$ is called from objects of class $Base$, it invokes $Base :: vm()$ to modify a . One callsite cannot precisely represent both cases. This approach also computes spurious data dependence: the approach is equivalent to using several objects, each belonging to a different type, to represent a polymorphic object. The data dependence graph construction algorithm cannot distinguish data members with the same name in these different objects.

A polymorphic object can also be used as a parameter or a data member for another object. For example, if class $Derived$ is derived from class $Base$, as in Figure 3, ba in $C(Base \&ba)$ becomes a polymorphic object. No previous work has addressed this issue.

Inheritance, a technique to facilitate code reuse, lets a derived class inherit data members and methods from its par-

```

class Derived
:public Base
{
    long d;
    vm(){
        d=d+b;
    }
public:
    Derived():Base(){
        d=0;
    }
}

m3() {
    d=d+1;
    m2(1);
}

m4() {
    m1();
}
}; // end of class

main2(){
    int i;
    Base *p;
    cin>>i;
    if(i>0)
        p=new Base();
    else
        p=new Derived();
    C(*p);
    (*p).m1();
}

```

Figure 3. Program 3.

ent classes, and define its own data members and methods. Existing techniques reuse the representation of base classes in the construction of the representation for a derived class, so as to construct the representation only for those methods defined in the derived class. This simple reuse strategy, however, will not work in the presence of virtual methods. For example, we cannot directly reuse the representation for $m1()$ in class $Base$ (Figure 2) when we construct the representation for class $Derived$ (Figure 3): in class $Base$, the callsite to virtual method $vm()$ is connected to $Base :: vm()$, which uses $Base :: a$ and $Base :: b$ and modifies $Base :: a$, whereas, in class $Derived$, the callsite to virtual method $vm()$ is connected to $Derived :: vm()$, which uses $Derived :: d$ and $Base :: b$ and modifies $Derived :: d$. Because the data members used by the two versions of $vm()$ differ, different callsite representations are required for the callsite to $vm()$ in $m1()$'s representation for class $Derived$ than in the callsite to $vm()$ in $m1()$'s representation for $Base$. Moreover, the callsite for $vm()$ in $Derived$ causes the header of $m1()$ in $Derived$'s SDG to differ from the header of $m1()$ in $Base$'s SDG: $m1()$ in class $Base$ has formal-in data member vertices for $Base :: a$ and $Base :: b$, and formal-out data member vertex for $Base :: b$; whereas, $m1()$ in class $Derived$ has formal-in data member vertices for $Derived :: d$ and $Base :: b$, and formal-out data member vertex for $Derived :: d$.

4 System Dependence Graph Extensions

This section presents an extended SDG that handles the problems discussed in the previous section. We base our discussion on a subset of C++ without exceptions; the techniques can also be applied to programs in other statically-typed object-oriented languages, such as Java. Moreover, we assume that all dereferences to pointers in the program have been resolved with an alias analysis algorithm (e.g., [13]). We further assume that data members of an object can be accessed only through a method (we replace a direct reference to a data member of an object as a method call). Finally, we treat static data members as global variables and static methods as global procedures because they do not as-

sociate with any particular runtime object.

Objects can be created and destroyed dynamically, and the number of runtime objects can vary during a program’s execution. Chatterjee and Ryder [3] group runtime objects into a finite number of equivalence classes; during static analysis, an equivalence class is represented by one object. They suggest that a possible way to create equivalence classes is to group all objects created at the same program point into an equivalence class. In this paper, we follow that suggestion, and use one object to represent the set of objects created by the same program statement.

Objects at Callsites. To represent a receiver object of a message, we follow Larsen and Harrold’s scheme. In our representation, a callsite contains *actual-in data member* vertices to represent data members that are referenced, and *actual-out data member* vertices to represent data members that are modified, by the called method. To support this representation, a *method entry* vertex, which is analogous to the entry vertex for a procedure, also contains *formal-in data member* vertices to represent those data members referenced in the method and *formal-out data member* vertices to represent those data members modified in the method. Like other parameter vertices, the actual-in/-out data member vertices are control dependent on the method call vertex, and the formal-in/-out data member vertices are control dependent on the method entry vertex. Formal-in data member vertices are labeled with assignments that copy the value of data members from an object into the scope of the method at the beginning of the method, and formal-out data member vertices are labeled with assignments that copy the value of data members from the scope of the method into the object at the end of the method. Under this approach, the data dependences related to data members of the class in a method are computed in the same way as the data dependences for a procedure.

Parameter Objects. To obtain more precision when an object is used as a parameter (*parameter object*), our SDG explicitly represents the data members of the object. A parameter object is represented as a tree: the root of the tree represents the object itself; the children of the root represent the object’s data members; and the edges of the tree represent the data dependences between the object and its data members. Under this representation, if a data member of the object is another object, we can further expand this data member into a subtree. Because of the existence of recursive class definitions, however, the tree may not always be able to be repeatedly expanded until all leaves are basic data types. One solution is to use k-limiting (i.e., only expand the tree to level k). In this paper, we use 1-limiting.

To compute more precise data dependences, when a tree is used to represent an object, uses or definitions are as-

sociated with the data-member vertices. If the object represented by the tree is used, then every data-member vertex in the tree is associated with a use of the data member; if the object represented by the tree is defined, every data-member vertex in the tree is associated with a definition of the data member. The algorithm in Section 5 shows how dependences related to a parameter object are computed.

The callsite $C(o)$ in function $D()$ in Figure 4 illustrates the representation of parameter objects at a callsite. At the callsite, the actual parameter o is represented as a tree, and the leaves of the tree represent o ’s data member a and b . The data-flow edges and summary edges involving o are connected among data member vertices. This representation helps the slicer exclude the data member vertex representing a at callsite $o.Base()$.

Polymorphic Objects. In a program written in a statically-typed object-oriented language, such as C++, the possible types of a polymorphic object can be determined statically and usually consist of a small set of such types [3]. A polymorphic object is represented as a tree: the root of the tree represents the polymorphic object and the children of the root represent objects of the possible types. When the polymorphic object is used as a parameter, the children are further expanded into trees; when the polymorphic object receives a message, the children are further expanded into callsites. Note that, in the latter case, our technique differs from the polymorphic choice scheme [10]: in our representation, we have one callsite for each possible object type; in the polymorphic choice scheme, different callsites are used only for different implementations of a virtual method. Our representation solve the two problems concerning polymorphic choice scheme that we described in Section 3.

Figure 4 shows the representation for function $C(Base \&ba)$ and $D()$ after we introduce class *Derived* and function $main2()$ of Figure 3. Because both $D()$ and $main2()$ call $C(Base \&)$ and present actual parameters of different types to $C(Base \&)$, the formal parameter ba in $C(Base \&ba)$ is a polymorphic parameter object; thus, we use trees to represent difference references to ba . The data dependence associated with data members occurs only between callsites with the same object type. For example, the actual-out vertex labeled with b at callsite $ba.m1() \rightarrow Derived$ is connected only to the actual-in vertex labeled with b at callsite $ba.m2(1) \rightarrow Derived$ — it is not connected to the actual-in vertex labeled with b at callsite $ba.m2(1) \rightarrow Base$, although these two callsites represent the same statement.

Classes in a Hierarchy. Within a class hierarchy, we try to maintain one copy of the representation for a method and this representation can be shared by different classes in the hierarchy. A class entry vertex groups the methods belong-

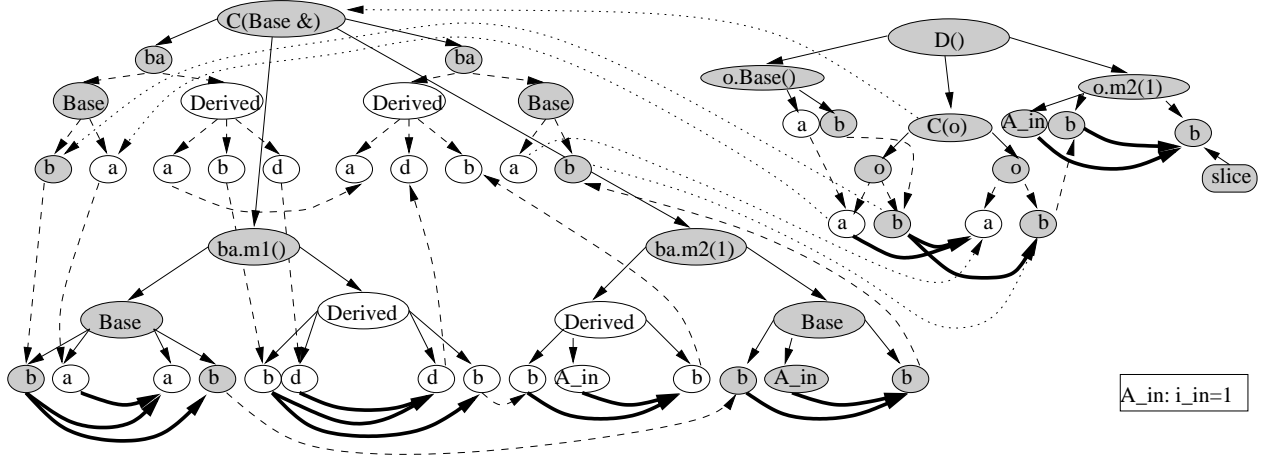


Figure 4. Representation for $C(Base \&)$ and $D()$ in SDG.

ing to one class together using membership edges. To identify the original class in which a method is declared, we prefix a method name with the name of the original class. As we have discussed before, however, a method might need a new representation when we construct the program dependence graph for a new class to the hierarchy. A method needs a new representation if (1) the method is declared in the new class, or (2) the method is declared in a lower level class in the hierarchy and calls a newly redefined virtual method directly or indirectly. For example, methods declared in *Derived* need a new representation because these methods satisfy (1). $Base :: m1()$ also needs a new representation because it satisfies (2): $Base :: m1()$ calls $Derived :: vm()$ which is redefined in class *Derived*.

5 SDG Construction Algorithm

`ConstructSDG` (Figure 5) inputs the control flow graph for each procedure or method in a program P , and outputs P 's SDG. `ConstructSDG` first builds procedure dependence graphs for methods in each class C , in a bottom-up fashion according to the class hierarchy (lines 1-17). (Procedures that do not belong to any class are processed by assuming that they belong to a dummy class). To compute procedure dependence graphs for C 's methods, `ConstructSDG` first identifies the methods that require new procedure dependence graphs (line 2). Then, for each method m that is declared in C , the algorithm constructs m 's procedure dependence graph (lines 3-8). For a method m declared in a base class, if m directly or indirectly calls a virtual method redefined in C , the algorithm constructs the procedure dependence graph from m 's procedure dependence graph that was built for the base class (lines 10-12); if m does not directly or indirectly call any virtual method redefined in C , the algorithm reuses m 's procedure dependence graph with the base class (line 14).

```

algorithm ConstructSDG
input Control flow graph (CFG) of Program  $P$ 
output System dependence graph (SDG) of  $P$ 
declare
begin ConstructSDG
1. foreach class  $C$ 
2.   Identify the methods that need new representations
3.   foreach method  $m$  declared in  $C$ 
4.     Compute control dependences for  $m$ 
5.     Preprocess  $m$  with usual data-flow analysis
6.     Expand objects in  $m$ 
7.     Compute data dependences for objects
8.   endfor
9.   foreach method  $m$  in the base classes
10.    if  $m$  is "marked" then
11.      Copy old procedure dependence graph
12.      Adjust callsites
13.    else
14.      Reuse  $m$ 's old procedure dependence graph
15.    endif
16.   endfor
17. endfor
18. Connect()
19. BuildSummaryEdges()
end ConstructSDG

```

Figure 5. Algorithm to construct SDG for P .

After constructing the procedure dependence graph for each method, `ConstructSDG` calls `Connect()` (line 18) to connect, with binding edges, every callsite in these procedure dependence graphs with the entry of the called method. The final step in the construction of an SDG is the computation of the summary edges at callsites; procedure `BuildSummaryEdges()` can be implemented for this purpose using an existing technique (e.g., in [11]).

5.1 Identify Methods Requiring New Procedure Dependence Graphs

`ConstructSDG` uses class call graphs to identify methods that need new procedure dependence graphs (line

2). Our class call graph differs from previous definitions in that it distinguishes two types of call statements that can be present in an object-oriented program: 1) a message-sending statement — a call statement whose execution causes program control to transfer to another object; 2) a procedure-call statement — a call statement whose execution does not cause program control to transfer to another object. To define the class call graph, we consider only procedure-call statements and the relation induced by them; we ignore all message-sending statements because they cause control to transfer out of the object. A method m_1 has a *procedure-call relation* with another method m_2 if a procedure-call statement in m_1 calls m_2 when the procedure-call statement is executed.

When a new class is declared, ConstructSDG uses a marking procedure on the class call graph to identify the methods that need new procedure dependence graphs: first, it marks the methods declared in the class; then, if the class is derived from some base classes, it marks the methods in the base classes that can reach these marked methods by performing a backward traversal on the class call graph from these marked methods. All marked methods need new procedure dependence graphs.

ConstructSDG also uses the class call graph to compute the interfaces for the methods that need new procedure dependence graphs. The interface of a method or procedure m is defined by the set of non-local variables (i.e., parameters, global variables, or data members) to which m refers, denoted as $GREF(m)$, and the set of non-local variables that m might modify, denoted as $GMOD(m)$. $GMOD(m)$ and $GREF(m)$ can be computed by an existing side-effect analysis algorithm [9].

5.2 Construct Procedure Dependence Graphs for Methods in a New Class

ConstructSDG constructs the procedure dependence graph for a method declared in a new class from the method’s control flow graph (lines 4-7). Object-oriented languages and imperative languages share many constructs. Thus, ConstructSDG builds *control dependence graphs* for a method with an existing algorithm (line 4). ConstructSDG builds the *data dependence graph* for a method in three steps (lines 5-7). In the first step (line 5), the algorithm treats an object as a simple variable and preprocesses the method using a modified version of the usual data dependence graph construction algorithm [2]. In the second step (line 6), the algorithm expands a vertex that references an object into the representations discussed in Section 4. In the third step (line 7), the algorithm computes the data dependence for data members of the objects. These three steps handle objects at callsites, objects as parameters, and polymorphic objects.

Preprocess Methods. In this step, ConstructSDG uses a modified data dependence graph construction algorithm to preprocess a method. The algorithm creates a complete callsite for a procedure-call statement based on the $GMOD$ and $GREF$ of the called method using a technique that is similar to the construction of a callsite for procedure, which is described in Section 2. However, for a message-sending statement, the algorithm creates actual parameter vertices only for parameters and global variables in the callee’s $GREF$ and $GMOD$. To estimate the data dependences introduced by an object’s data members, the algorithm associates a use of the object with the call vertex if the message-sending statement is not a construction; similarly, the algorithm associates a definition of the object with the call vertex if the message-sending statement is not a destruction. If the message’s receiver has multiple types, then the algorithm creates a callsite for each possible type and group all these callsites with a polymorphic object vertex. In this case, however, the use and the definition of the object are associated with the polymorphic object vertex instead of the call vertices. When an object is used as a parameter, ConstructSDG treats the object as a simple variable and creates a vertex for it.

The inset graph in Figure 6 shows the result of the preprocessing of procedure $C(Base \ \&)$ (in Figure 2). The algorithm does not create actual parameter vertices in this graph for ba ’s data members at callsites representing message-sending statement “ $ba.m1()$,” and “ $ba.m2(1)$.” The algorithm associates use and definition of ba with the call vertices labeled with $ba.m1()$ and $ba.m2(1)$. Thus, there is a data dependence edge between these two vertices.

Expand Objects. In the second step, ConstructSDG expands vertices that reference an object using the tree-representation technique discussed in Section 4. This step uses an *object-flow subgraph*: a subgraph in the data dependence graph of a method or procedure. The subgraph includes the vertices that reference O (i.e., method-sending call vertices and object parameter vertices) and all the data dependence edges built by ConstructSDG in the first step among these vertices. The bold vertices and edges in the inset graph in Figure 6 shows the object-flow subgraph for object ba in procedure $C(Base \ \&)$.

ConstructSDG considers each vertex v on an object-flow subgraph. If v is a parameter vertex representing a single-typed object, then the algorithm expands it into a tree. If v is a parameter vertex representing a polymorphic object, then the algorithm first creates a child vertex for each possible object type; and then it expands each child vertex into a tree. If v is a call vertex, then the algorithm creates the actual parameter vertices for the data members in the callee’s $GREF$ and $GMOD$. If v is a vertex

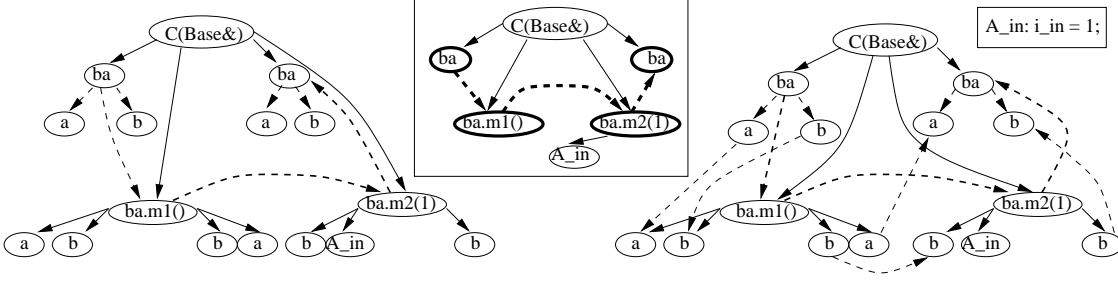


Figure 6. Construction of the PDG for $C(\text{Base \&})$

representing a call to a polymorphic object, then for each callsite connected to v , the algorithm creates the actual parameter vertices for data members in the callee’s *GREF* and *GMOD* at the callsite. The left graph in Figure 6 shows the procedure dependence graph for $C()$ (in Figure 2) after the vertices in ba ’s object-flow subgraph are expanded.

Build Data Dependences for Data Members. After expanding the vertices in O ’s object-flow subgraph, *ConstructSDG* propagates the definitions of data members along the object-flow subgraph and builds the data dependence for the data members. Given a callsite c on the object-flow subgraph, we represent the definition set of data members with the set of actual-out data member vertices associated with c . Similarly, we represent the use set of c with the set of actual-in data member vertices. For an object parameter vertex, if the vertex defines the object, then we represent the definition set of the data members with the set of its data member vertices; otherwise, if the vertex uses the object, then we represent the use set of the data members with the set of its data member vertices. The propagation of definitions can be done in a usual way. After the propagation, at each vertex on the object-flow subgraph, if there is a use of the data member d , then for each reaching definition of d , we create a data dependence edge.

5.3 Construct Procedure Dependence Graphs for Methods in the Base Class

ConstructSDG builds a new procedure dependence graph for a method m declared in a base class if m has been identified to directly or indirectly call a virtual method redefined by the derived class. *ConstructSDG* reuses the control dependence information and the data dependence information from m ’s procedure dependence graph built for the base class. Modifications are needed only at procedure callsites that call methods whose *GREF* and *GMOD* have changed: the algorithm might need to add or remove some actual-in vertices or actual-out vertices. An actual-in vertex representing N has a use of N associated with it. Similarly, an actual-out vertex representing N is associated with a definition of N . Thus, adding or removing an actual-in

or actual-out vertex requires the algorithm to recompute the data dependences related to the use or the definition. Notice that because the formal parameters in the callee’s *GREF* and *GMOD* do not change, the modifications only apply to data members or global variables.

6 Slicing an Object

In an object-oriented program, when an object is instantiated from a class, the data members and methods of the class are instantiated for the object. Because we use only one representation for all instances of a method, however, the set of statements in a method identified by a slicing algorithm, such as Horwitz et al.’s [7], includes statements in different instances of the method. For tasks, such as debugging and program understanding, we would like to focus our attention on one object at a time. To do this, we want to identify the statements in the methods of a particular object that might affect the slicing criterion — we call this *object slicing*.

To slice an object O for a criterion $\langle v, p \rangle$, we first obtain the complete slice from the SDG for $\langle v, p \rangle$. Then, we identify O ’s message-sending callsites, which were marked by the slicing algorithm; the methods invoked by these callsites are the only methods of O that can affect $\langle v, p \rangle$. Next, we identify the statements in these methods that are included in the slice because of the invocations at the selected callsites; we call these statements O ’s *slice*.

To identify O ’s message-sending callsites, which were marked by the slicing algorithm, we examine the vertices on O ’s object flow subgraph in the method or procedure in which O is constructed. If O is passed as a parameter to another method or procedure, we traverse the binding edge at the callsite, and continue to examine the object flow subgraph of the formal parameter object in the called method or procedure. The traversal continues until all callsites of the object have been examined.

A statement S in an O ’s method m can affect $\langle v, p \rangle$ in several ways. First, S can affect an actual-out parameter A_{out} at O ’s message sending callsite that calls m , and A_{out} in turn can affect $\langle v, p \rangle$. This case occurs when the execution of p could occur after the return of the callsite. The

set of A_{out} 's is the set of actual-out vertices that are marked by the two-phase slicing algorithm when we compute the slice for the entire program. Given such an A_{out} , S is a statement that can reach F_{out} , the formal-out vertex that is bound to A_{out} , through a path consisting only of vertices in m . We identify this type of statement using a backward traversal within m , beginning at F_{out} .

Second, if p is also in m , S can affect $\langle v, p \rangle$ through a path consisting of only the vertices within m . We identify this type of statement using a backward traversal within m , beginning at p .

Third, S can affect $\langle v, p \rangle$ by affecting an actual-in parameter A_{in} at a message sending callsite c in m , in which case A_{in} can reach p by some path that does not traverse any parameter-out edge. This means that p is executed before the return of callsite c . To identify this type of statement we must first mark the set of A_{in} 's. This marking can be done by a backward traversal from p without traversing any parameter-out edges. The marking can also be accomplished by adding a mark to the actual-in parameter vertices when we perform the first phase of the slicing algorithm. In this case, the set of A_{in} would be the A_{in} 's with the extra mark at the callsites within m . After we mark the set of A_{in} 's, we identify the set of S with a backward traversal within m , beginning at the marked actual-in parameter vertices.

It is easy to see that if a statement in an object's method can affect $\langle v, p \rangle$, then it must be marked by one of the three cases. Therefore, the above process can be used to compute O 's slice.

In the above discussion, we assume that, when a method is invoked to handle a message, the method will not call other methods. If this assumption does not hold, the search for the statement S may require a backward traversal through several methods. To handle this situation, when we identify O 's message sending callsite, we also mark the methods that might be involved in handling the message. We mark these methods with a forward traversal beginning from the called method on the class call graph. The marked methods are the only methods of O that could affect $\langle v, p \rangle$.

To present the way in which our algorithm handles this situation, we distinguish parameter binding edges associated with procedure callsites from parameter binding edges associated with message-sending callsites: parameter binding edges associated with a procedure callsite are called *intra-object binding edges*; parameter binding edges associated with a message sending callsite are called *inter-object binding edges*. Thus, we have *intra-object parameter-in edges*, *intra-object parameter-out edges*, *inter-object parameter-in edges*, and *inter-object parameter-out edges*. Inter-object binding edges represent the boundaries of objects.

To search for S that might affect the slicing criterion, we use a two-phase algorithm. As discussed in previous paragraphs, the backward traversal begins with a formal-out parameter vertex (in case 1), the slicing criterion (in case 2), or an actual-in parameter vertex (in case 3). In the first phase, the algorithm traverses backwards only through data dependence edges, control dependence edges, summary edges, and intra-object parameter-in edges where both the caller and callee have been marked on the class call graph. In the second phase, beginning from vertices marked in the first phase, the algorithm traverses backwards through all edges except inter-object binding edges and intra-object parameter-out edges. Vertices reached by the traversal are those in O 's slice.

Figure 7 shows a part of the SDG for Program 2. To obtain o 's slice on statement "b=b+i;" in $Base :: m2()$, the algorithm first computes the complete slice (shaded vertices on the SDG). Then, the algorithm identifies o 's message-sending callsites $o.m2(1)$ and $o.Base()$, which have been marked by the slicer. The algorithm concludes that only the statements in $Base :: Base()$ and $Base :: m2()$ can affect the execution of the slicing criteria through o . To identify the statements in $Base :: Base()$, the algorithm performs a backward traversal from the formal-out parameter $Base() \rightarrow F2_{out}$ and marks $Base() \rightarrow F2_{out}$ and "b=0;". To identify the statements in $Base :: m2()$, the algorithm performs a backward traversal from the slicing criteria, without going through any inter-object binding edge. The algorithm marks vertices "b=b+i;", $m2() \rightarrow F2_{in}$, and $m2() \rightarrow F3_{in}$. The shaded vertices with a dotted boundary depict those vertices identified by the algorithm. From the graph, we can see that, although the slice obtained by the usual slicing algorithm includes statements in method $m1()$, o 's slice omits these statements because these statements in $m1()$'s instance for o do not affect the slicing criterion.

At first glance, slicing an object may seem like an expensive process because we must obtain the complete slice for the program. However, if we use object slicing to browse slices in an object-oriented program, then the technique for object slicing is quite efficient: we need only slice the entire program once; then to slice an object, we need only traverse the portion of the system dependence graph related to the object being considered.

7 Conclusion

We have presented an approach to extend the SDG to represent object-oriented programs. Compared to other existing approaches, our representation support more precise slicing: it distinguishes data members belonging to different objects; it represents data members even when objects are used as parameters; it considers the effect of polymorphism on callsites and parameters. We have also presented an ef-

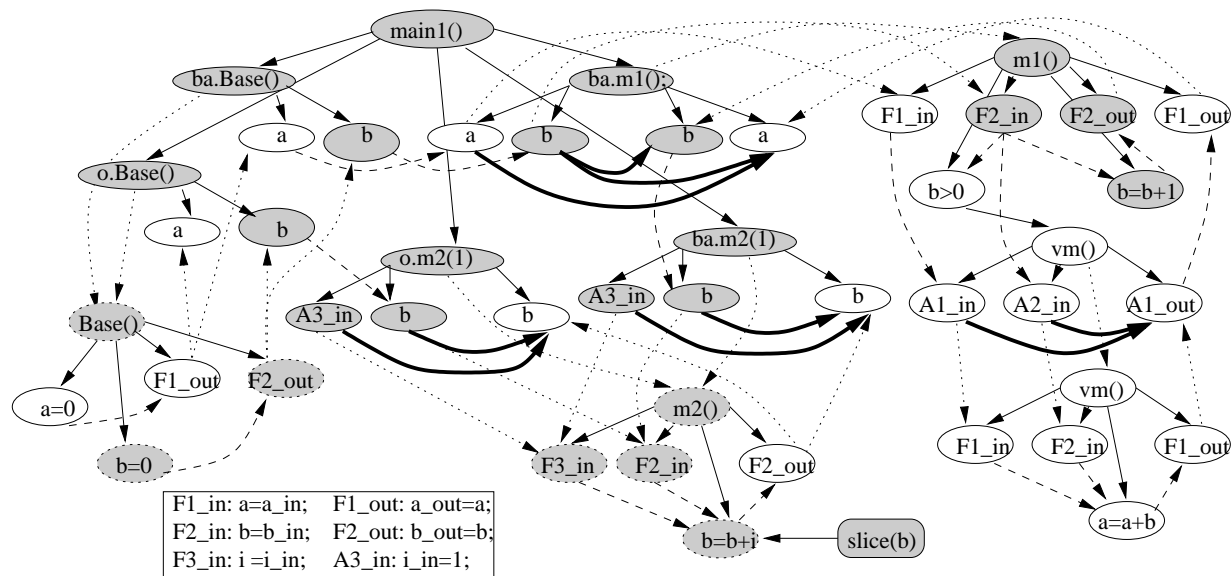


Figure 7. The system dependence graph for Program 2.

efficient algorithm to construct such an SDG for an object-oriented program. Finally, we have presented a new concept, object slicing, which enables the user to inspect the effects of a particular object on the slicing criteria. Object slicing provides better support for debugging and program understanding for large scale programs.

In our work, we also provide efficient mechanism to represent classes in class libraries and to represent incomplete program. Because of the space limitations, however, we have not presented these techniques in this paper.

Our future work includes implementing our SDG construction algorithm, and assessing the effectiveness and efficiency of using the SDG to represent object-oriented programs and of static slicing on object-oriented program using the two-phase slicing algorithm.

8 Acknowledgments

This work was supported in part by a grant from Microsoft Inc., by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University.

References

- [1] H. Agrawal. On slicing programs with jump statements. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 60–73, June 1994.
- [2] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 1986.
- [3] R. Chatterjee and B. Ryder. Scalable, flow-sensitive type inference for statically typed object-oriented languages.

- Technical Report DCS-TR-326, Rutgers University, August 1994.
- [4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [5] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.
- [6] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of Object-Oriented Prog. Sys., Lang., and Appl.*, pages 108–124, Oct. 1997.
- [7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [8] D. Jackson and E. J. Rollins. A new model of program dependence for reverse engineering. *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering*, pages 2–10, December 1994.
- [9] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.
- [10] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *18th International Conference on Software Engineering*, pages 495–505, Mar. 1996.
- [11] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *Proc. of Second ACM Conf. on Foundations of Softw. Eng.*, pages 11–20, Dec. 1994.
- [12] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.
- [13] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *19th International Conference on Software Engineering*, pages 433–443, May 1997.