Slicing Object-Oriented Software[*]

Loren Larsen[†] and Mary Jean Harrold[‡]
Department of Computer Science
Clemson University

Abstract

We describe the construction of system dependence graphs for object-oriented software on which efficient slicing algorithms can be applied. We construct these system dependence graphs for individual classes, groups of interacting classes, and complete object-oriented programs. For an incomplete system consisting of a single class or a number of interacting classes, we construct a procedure dependence graph that simulates all possible calls to public methods in the class. For a complete system, we construct a procedure dependence graph from the main program in the system. Using these system dependence graphs, we show how to compute slices for individual classes, groups of interacting classes and complete programs. One advantage of our approach is that the system dependence graphs can be constructed incrementally because representations of classes can be reused. Another advantage of our approach is that slices can be computed for incomplete object-oriented programs such as classes or class libraries. We present our results for C^{++}, but our techniques apply to other statically typed object-oriented languages such as Ada-95.

1 Introduction

Program slicing has many applications such as debugging, code understanding, program testing, reverse engineering, and metrics analysis[4, 5, 6, 11, 15, 25, 26]. Weiser[27] defines a *slice* with respect to a slicing criterion that consists of a program point p and a subset of program variables V. His slices are executable programs that are constructed by removing zero or more statements from the original program. His algorithm uses dataflow analysis on control flow graphs to compute intraprocedural and interprocedural slices. Ottenstein and Ottenstein[20] define a slicing criterion to consist of a program point p and a variable v that is defined or used at p. They use a graph reachability algorithm on a program dependence graph to compute a slice that consists of all statements and predicates of the program that may affect the value of v at p. Horwitz, Reps and Binkley[14] also use dependence graphs to compute slices. They developed an interprocedural program representation, the system dependence graph, and a two-pass graph reachability slicing algorithm that uses the system dependence graph. Their two-pass algorithm computes more precise interprocedural slices than previous algorithms because it uses summary information at call sites to account for the calling context of called procedures. Researchers have extended language features represented by system dependence graphs[2, 3, 8, 9, 13, 16], and proposed variations of dependence graphs that facilitate finer-grained slices[15, 18]. Researchers have also considered ways to represent object-oriented programs[12, 19, 24]. However, no existing techniques adequately define system dependence graphs and slicing algorithms for the full range of object-oriented features.

Object-oriented features, such as classes and objects, inheritance, polymorphism and dynamic binding, and scoping, impact the development of an efficient object-oriented program representation. A *class* defines the attributes that an instance of that class (an object) will possess. A class's attributes consist of (1) *instance variables* that implement the object's state and (2) *methods* that implement the operations on the object's state. We often design, implement, and test classes without knowledge of calling environments. Thus, an efficient graph representation for object-oriented software should consist of a class representation that can be reused in the construction of other classes and applications that use the class.

Derived classes are defined through *inheritance*, which permits the derived class to inherit attributes from its parent classes, and extend, restrict, redefine, or replace them in some way. Just as inheritance facilitates code reuse, an efficient graph representation for a derived class should facilitate the reuse of analysis information. Construction of the representation for a derived class should reuse analysis information computed for base classes, and only compute information that is defined in the derived class.

Polymorphism is an important feature of object-oriented languages that 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, unless the type can be determined statically.

[*]This work was partially supported by grants from Microsoft, Inc. and Data General Corp., and by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University.

[†]Current address: IBM Corporation, Research Triangle Park, NC, llarsen@raleigh.ibm.com.

[‡]Current address: Department of Computer and Information Science, The Ohio State University, 395 Dreese Lab, 2015 Neil Avenue, Columbus OH, 43210, harrold@cis.ohio-state.edu.

Although the visibility of an object's instance variables (state) is limited, these instance variables retain their values between method calls to the object. A representation must account for the dependencies of instance variables between calls to the object's methods by a calling program even though the instance variables are not visible to the calling program.

To address these object-oriented features, we developed system dependence graphs for object-oriented software on which an efficient interprocedural slicing algorithm[14, 23] can be applied. We construct these system dependence graphs for individual classes, groups of interacting classes, and complete object-oriented programs. For each class C in the system, we construct a class dependence graph that we reuse in the construction of classes that are derived from C and classes that instantiate C. For an incomplete system consisting of a single class or a number of interacting classes, we construct a procedure dependence graph that simulates all possible calls to public methods in the class. For a complete object-oriented system, we construct a procedure dependence graph from the main program in the system.

The main contribution of this work is a representation for single classes, interacting classes, and complete object-oriented programs on which slices can be computed efficiently. Our approach permits the computation of slices for individual classes. Our techniques focus on efficiency in construction and storage by reusing previously computed components of the representation whenever possible. We present our results for C^{++} but our techniques can be applied to other statically typed object-oriented programming languages such as Ada-95.

In the next section, we overview interprocedural slicing using the system dependence graph. Section 3 presents our system dependence graphs for object-oriented software without considering issues such as aliasing, arrays, reference parameters, and finer-grained representations since existing techniques for handling these features are applicable to our graphs. Although these issues are relevant to the topic of program representation, they introduce no issues unique to object-oriented software. In Section 4, we discuss the computation of backward static slices on our system dependence graphs; forward slices can be computed similarly. Section 5 discusses the space requirements for our graphs, and Section 6 presents our conclusions and future work.

2 Background

A *system dependence graph* (SDG)[14] is a collection of procedure dependence graphs, one for each procedure. A *procedure dependence graph*[10] 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, and *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-in* and *formal-out* vertices. An SDG contains a formal-in vertex for each formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified[17] by the procedure. An SDG associates each call site in a procedure with a *call* vertex and a set of *actual-in* and *actual-out* vertices. An SDG contains an actual-in vertex for each actual parameter at the call site and an actual-out vertex for each actual parameter that may be modified by the called procedure. At procedure entry and call sites, global variables are treated as parameters. Thus, there are actual-in, actual-out, formal-in and formal-out vertices for these global variables. SDGs connect procedure dependence graphs at call sites. A *call* edge connects a procedure call vertex to the entry vertex of the called procedure's dependence graph. Parameter-in and parameter-out edges represent parameter passing. *Parameter-in* edges connect actual-in and formal-in vertices, and *parameter-out* edges connect formal-out and actual-out vertices.

Figure 1 shows a program `main` and its SDG. In the figure, circles represent program statements; they are labeled by statement numbers. Ellipses represent parameter vertices; they are labeled with A*i*_in or A*i*_out for actual parameters and F*i*_in or F*i*_out for formal parameters. The key in the figure describes the labels associated with each parameter vertex. 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, we use C7→A1_in to refer to the parameter vertex representing actual parameter "a_in=current_floor" in the call to **add()** at C7, and we use C9→A1_in to refer to actual parameter "a_in=current_floor" in the call to **add()** at C9.

In the figure, solid edges represent control dependencies, dashed edges represent data dependencies, and dotted edges represent procedure calls and parameter bindings. For example, the **while** statements in S6 and S8 are control dependent on the value of the predicate in S5. Thus, there are control dependence edges (S5, S6) and (S5, S8) in the SDG. The value of **current_floor** in S2 may be used in statements S6 and S8, and **current_floor** appears as an actual parameter at call sites at C7 and C9. Thus, there are data dependence edges (S2, S6), (S2, S8), (S2, C7→A1_in) and (S2, C9→A1_in). A parameter binding occurs at the call to **add()** at C7 between **current_floor** in **main** and ***a** in **add()**; a similar binding occurs between **current_floor** and ***a** at the call to **add()** at C9. These bindings result in parameter-in edges (C7→A1_in, E11→F1_in) (C9→A1_in, E11→F1_in), and parameter-out edges (E11→F1_out, C7→A1_out) and (E11→F1_out, C9→A1_out).

Horwitz, Reps and Binkley[14] compute interprocedural slices by solving a graph reachability problem on an SDG. To obtain precise slices, the computation of a slice must preserve the calling context of called procedures, and ensure that only paths corresponding to legal call/return sequences are considered. To facilitate the computation of interprocedural slicing that considers the calling context, an SDG represents

```
E0: main() {
      int current_floor;
      int top_floor:
      int current_direction;
S1:   int floor = 5;
S2:   current_floor=1;
S3:   top_floor = 10;
S4:   current_direction = UP;
S5:   if (current_direction == UP)
S6:      while ((current_floor != floor) &&
                (current_floor <= top_floor))
C7:         add(&current_floor, 1);
      else
S8:      while ((current_floor != floor) &&
                (current_floor > 0))
C9:         add(&current_floor, -1);
S10:  printf("%d", current_floor);

    }
E11: add(int *a, int b) {
S12:    *a = *a + b;
    }
```

Key for Parameter Vertices

A1_in:	a_in = current_floor
A2_in:	b_in = 1
A1_out:	current_floor = a_out
A3_in:	b_in = -1
F1_in:	a = a_in
F2_in:	b = b_in
F1_out:	a_out = a

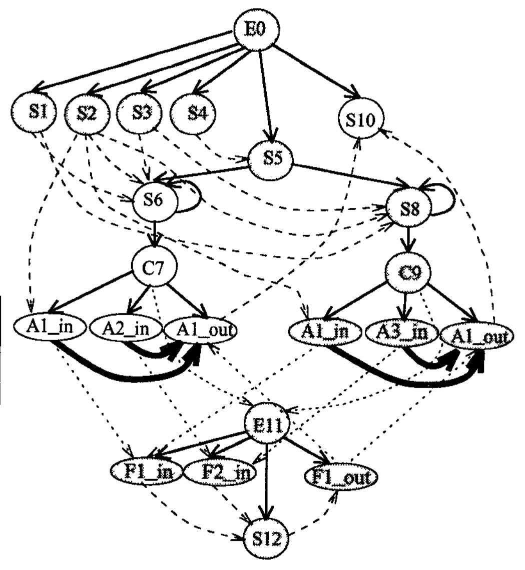


Figure 1: Example program `main` and its system dependence graph.

the flow of dependencies across call sites. A *transitive flow of dependence* occurs between an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence can be caused by data dependencies, control dependencies, or both. A *summary* edge models the transitive flow of dependence across a procedure call. In the SDG of Figure 1, bold lines represent summary edges. Thus, edges (C7→A1_in, C7→A1_out) and (C9→A1_in, C9→A1_out) represent the fact that in procedure `add()`, the value of `current_floor` that is passed to `add()` affects the value of `current_floor` that is returned by `add()`.

The first pass of the interprocedural slicing algorithm traverses backward along all edges except parameter-out edges, and marks those vertices reached. The second pass 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 vertices marked during pass one and pass two.

To illustrate, consider the computation of a slice for vertex C9→A1_out. During the first pass, the algorithm marks vertices C9→A1_out, C9, S8, S5, E0, S4, S3, S2, S1, C9→A3_in, and C9→A1_in. On the second pass, a traversal starts at each vertex reached during pass 1, and the algorithm marks vertices E11→F1_out, E11, S12, E11→F2_in, and E11→F1_in. The vertices in the slice are shaded in the SDG in Figure 1.

3 System Dependence Graphs

In this section, we describe our system dependence graphs for both incomplete and complete object-oriented software.

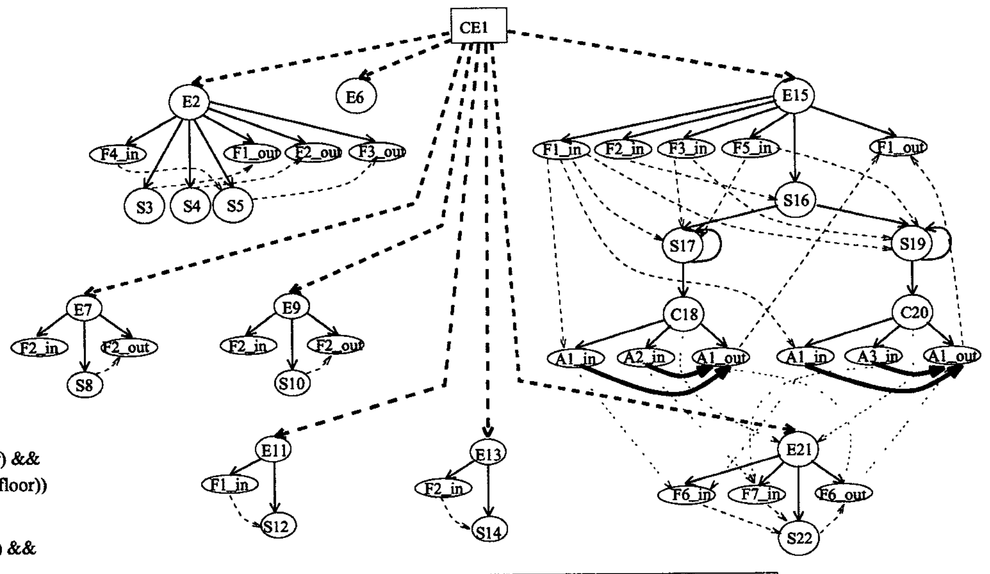
3.1 Class Dependence Graphs

This section describes the construction of class dependence graphs for single classes, derived classes and interacting classes. The section also discusses the way in which our graphs represent polymorphism.

Representing Base Classes

To facilitate analysis and reuse, we represent each class in a system by a *class dependence graph* (ClDG)[25]. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments. Each method in a ClDG is represented by a procedure dependence graph, which we described in Section 2. Each method has a *method entry* vertex that represents the entry into the method. A ClDG also contains a *class entry* vertex that is connected to the method entry vertex for each method in the class by a *class member* edge. Class entry vertices and class member edges let us quickly access method information when a class is combined with another class or system. Our ClDG construction expands each method entry by adding formal-in and formal-out vertices. We add formal-in vertices for each formal parameter in the method, and formal-out vertices for each formal reference parameter that is modified by the method. Additionally, we add formal-in and formal-out parameters for global variables that are referenced in a method. Finally, since a class's instance variables are accessible to all methods in the class, we treat them as globals to methods in the class, and we add formal-in and formal-out vertices for all instance variables that are referenced in the method. The exception to this representation for instance variables is that our construction omits formal-in vertices for instance variables in the class constructor and formal-out vertices for instance variables in the class destructor.

```
CE1:   class Elevator {
       public:
E2:      Elevator(int l_top_floor)
S3:        { current_floor = 1;
S4:          current_direction = UP;
S5:          top_floor = l_top_floor; }
E6:      virtual ~Elevator() {}
E7:      void up()
S8:        {current_direction = UP; }
E9:      void down()
S10:       { current_direction = DOWN; }
E11:     int which_floor()
S12:       { return current_floor; }
E13:     Direction direction()
S14:       { return current_direction; }
E15:     virtual void go (int floor)
S16:       { if (current_direction == UP)
S17:         { while (current_floor != floor) &&
                    (current_floor <= top_floor))
C18:           add(current_floor, 1); }
           else
S19:         { while (current_floor != floor) &&
                    (current_floor > 0))
C20:           add(current_floor, -1); }
         }
       private:
E21:     add(int &a, const int& b)
S22:       { a = a + b; };
       protected:
         int current_floor;
         Direction current_direction;
         int top_floor;
       };
```



Key for Parameter Vertices

F1_in:	current_floor = current_floor_in	F6_in:	a = a_in
F1_out:	current_floor_out = current_floor	F6_out:	a_out = a
F2_in:	current_direction = current_direction_in	F7_in:	b = b_in
F2_out:	current_direction_out = current_direction	A1_in:	a_in = current_floor
F3_in:	top_floor = top_floor_in	A1_out:	current_floor = a_out
F3_out:	top_floor_out = top_floor	A2_in:	b_in = 1
F4_in:	l_top_floor = l_top_floor_in;	A3_in:	b_in = -1
F5_in:	floor = floor_in		

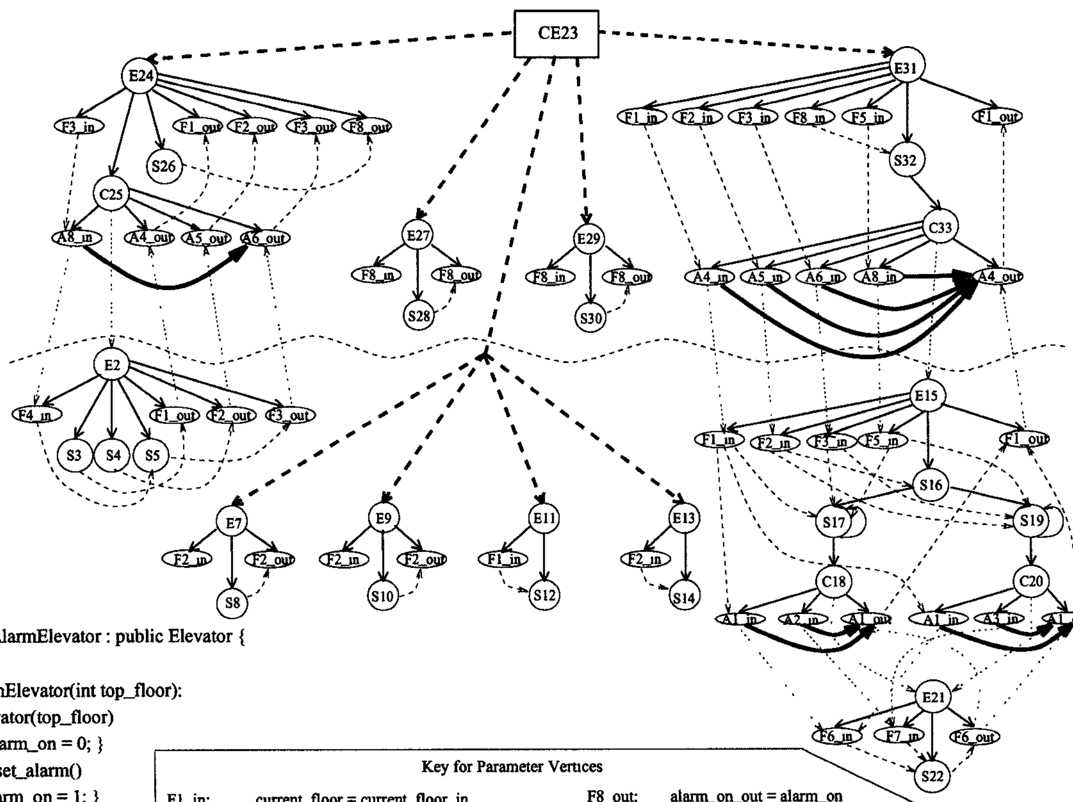
Figure 2: A C^{++} **Elevator** class and its ClDG.

Figure 2 shows the ClDG for a C^{++} `Elevator` class; the `go()` method is similar to the C program in Figure 1. A rectangle represents the class entry vertex and is labeled by the statement label associated with the class entry. Circles represent statements, and are labeled with the corresponding statement number in the class. For example, CE1 is the class entry vertex, and E2, E6, E7, E9, E11, E13, E15 and E21 are method entry vertices. Bold dashed edges represent class member edges that connect the class entry vertex to each method entry vertex; (CE1, E2), (CE1, E6), (CE1, E7), (CE1, E9), (CE1, E11), (CE1, E13), (CE1, E15) and (CE1, E21) are class member edges. Each method entry vertex is the root of a subgraph that is itself a partial SDG containing control dependence edges (shown as solid lines), data dependence edges (shown as light dashed lines), call and parameter edges (shown as dotted lines), and summary edges (shown as solid bold lines). Constructor method `Elevator()` has no formal-in vertices for the three instance variables, since these variables cannot be referenced before they are allocated by the class constructor. The ClDG for `Elevator` represents only necessary parameter information. For example, methods `up()`, `down()` and `direction()` only reference instance variable `current_direction`. Thus, methods represented by method entries E7, E9 and E13 only contain the required formal-in and formal-out vertices for `current_direction`.

Since methods in a class can interact with each other or with other methods, a ClDG represents the effects of method calls by a *call* vertex. At each call vertex, there are actual-in and actual-out vertices to match the formal-in and formal-out vertices present at the entry to the called method. For example, in the ClDG of Figure 2, C18 and C20 represent calls to `add()`.

A ClDG for a C^{++} class must represent effects of **return** statements, which cause a method to return a value to its caller. In a ClDG, each **return** statement is connected to its corresponding call vertex using a parameter-out edge.[1] Additionally, for every actual-in parameter that may affect the returned value, summary edges are added between actual-in vertices and the call vertex; these summary edges facilitate interprocedural slicing. Figure 5 lists a program that calls method `which_floor()`, which contains a **return** statement, and illustrates the **return** statement's representation in the SDG. In the figure, parameter-out edge (S12, S39) connects **return** statement S12 to call vertex S39, and summary edge (A4_in, S39) indicates that the value of `current_floor` on entry to method `which_floor()` affects the value returned by `which_floor()`.

[1]This edge is similar to the return-link edge described in Reference [18].



```
CE23:  class AlarmElevator : public Elevator {
         public:
E24:       AlarmElevator(int top_floor):
C25:         Elevator(top_floor)
S26:           {alarm_on = 0; }
E27:       void set_alarm()
S28:         { alarm_on = 1; }
E29:       void reset_alarm()
S30:         { alarm_on = 0; }
E31:       void go(int floor)
S32:         { if (!alarm_on)
C33:             Elevator::go(floor)
             };
         protected:
           int alarm_on;
       };
```

Key for Parameter Vertices

F1_in:	current_floor = current_floor_in	F8_out:	alarm_on_out = alarm_on
F1_out:	current_floor_out = current_floor	A1_in:	a_in = current_floor
F2_in:	current_direction = current_direction_in	A1_out:	current_floor = a_out
F2_out:	current_direction_out = current_direction	A2_in:	b_in = 1
F3_in:	top_floor = top_floor_in	A3_in:	b_in = -1
F3_out:	top_floor_out = top_floor	A4_in:	current_floor_in = current_floor
F4_in:	l_top_floor = l_top_floor_in	A4_out:	current_floor = current_floor_out
F5_in:	floor = floor_in	A5_in:	current_direction_in = current_direction
F6_in:	a = a_in	A5_out:	current_direction = current_direction_out
F6_out:	a_out = a	A6_in:	top_floor_in = top_floor
F7_in:	b = b_in	A6_out:	top_floor = top_floor_out
F8_in:	alarm_on=alarm_on_in	A8_in:	l_top_floor_in = top_floor

Figure 3: A C^{++} class **AlarmElevator** and its ClDG.

Representing Derived Classes

We construct a ClDG for a derived class by constructing a representation for each method defined by the derived class, and reusing the representations of all methods that are inherited from base classes. We create a class entry vertex for the derived class, and add class member edges connecting this class entry vertex to the method entry vertex of each method in the definition of the derived class. We also create class member edges to connect the class entry vertex to the method entry vertices of any methods defined in the base class that are inherited by the derived class. Formal-in vertices for a method represent the method's formal parameters, instance variables in the derived class or base classes that may be referenced by a call to this method, and global variables that may be referenced by this method. Similarly, formal-out vertices for a method represent the method's formal parameters, instance variables in base or derived classes that may be modified by a call to this method, and global variables that may be modified by a call to this method.

Figure 3 shows a C^{++} class `AlarmElevator` that is derived from class `Elevator`; we omit reference to the destructor of this class in our discussion. CE23 is the entry vertex for class `AlarmElevator`. Class member edges from CE23 to the method entry of each method in the definition of `AlarmElevator` are (CE23, E24), (CE23, E27), (CE23, E29) and (CE23, E31). Since `AlarmElevator` inherits methods `up()`, `down()`, `which_floor()` and `direction()` from class `Elevator`, there are also class member edges (CE23, E7), (CE23, E9), (CE23, E11) and (CE23, E13), respectively. The constructor for `AlarmElevator` calls the constructor for `Elevator`. Thus, our ClDG construction connects call vertex C25 in **AlarmElevator()** to entry vertex E2 in **Elevator()** by call edge (C25, E2). Virtual method **go()** in **Elevator** is not directly accessed in **AlarmElevator**; **go()** is redefined in **AlarmElevator** and calls **Elevator::go()**. Thus, our ClDG construction connects call vertex C33 in **AlarmElevator::go()** to entry vertex E15 in **Elevator::go()**. For call sites C25 and C33, our ClDG construction adds parameter-in and parameter-out edges. Summary edges, which were computed for each method in **Elevator** when the ClDG for that class was created, are reflected to call sites from **AlarmElevator** to **Elevator**. For example, summary edge (C25→A8_in, C25→A6_out) represents the fact that the value of **l_top_floor** on entry to **Elevator()** affects the value of **top_floor** on exit from **Elevator()**. Our ClDG construction also adds summary edges where required at the call to **Elevator::go()** in **AlarmElevator::go()**. The portion of the ClDG below the wavy dashed line in the figure is reused from the **Elevator** class.

Representing Interacting Classes

In object-oriented software a class may instantiate another class either through a declaration or by using an operator such as **new**. For example, a class may instantiate the **Elevator** class with the statement **Elevator elevator_object(10)** or with the statement **elevator_pointer = new Elevator(10)**. When class $C1$ instantiates class $C2$, there is an implicit call to $C2$'s constructor. To represent this implicit constructor call, our ClDG construction adds a call vertex in $C1$ at the location of the instantiation. A call edge connects this call vertex to $C2$'s constructor method. Our ClDG construction also adds actual-in and actual-out vertices at the call vertex to match the formal-in and formal-out vertices in $C2$'s constructor. Figure 5 illustrates the representation of interacting classes. Statements S36 and S37 in **main()** instantiate objects of type **AlarmElevator** and **Elevator**, respectively. The call vertex for S36 has actual-in and actual-out vertices to match the formal-in and formal-out vertices associated with E24, the method entry vertex for **AlarmElevator()**. Likewise, the call vertex for S37 has actual-in and actual-out vertices to match the formal-in and formal-out vertices associated with E2, the method entry vertex for **Elevator()**.

When there is a call site in method $M1$ in $C1$ to method $M2$ in the public interface of $C2$, our construction adds a call edge between the call vertex in $C1$ and $M2$'s entry vertex; parameter edges are also added. The linkage of these two ClDGs forms a new ClDG that represents a partial system.

Representing Polymorphism

A ClDG must represent polymorphic method calls, which occur when a method call is made and the destination of the call is unknown at compile-time. A ClDG uses a *polymorphic choice* vertex to represent the dynamic choice among the possible destinations. A call vertex corresponding to a polymorphic call has a call edge incident to a polymorphic choice vertex. A polymorphic choice vertex has call edges incident to subgraphs that represent calls to each possible destination. The polymorphic choice vertex represents the dynamic selection of a destination. Static analysis, however, must consider all possibilities. In Figure 5, P1 is a polymorphic choice vertex that represents a dynamic choice between calls to **Elevator::go** and **AlarmElevator::go**. Algorithms for statically eliminating infeasible destinations of a polymorphic call are described in References [1, 21], but a precise solution to the type inferencing problem is NP-hard[21].

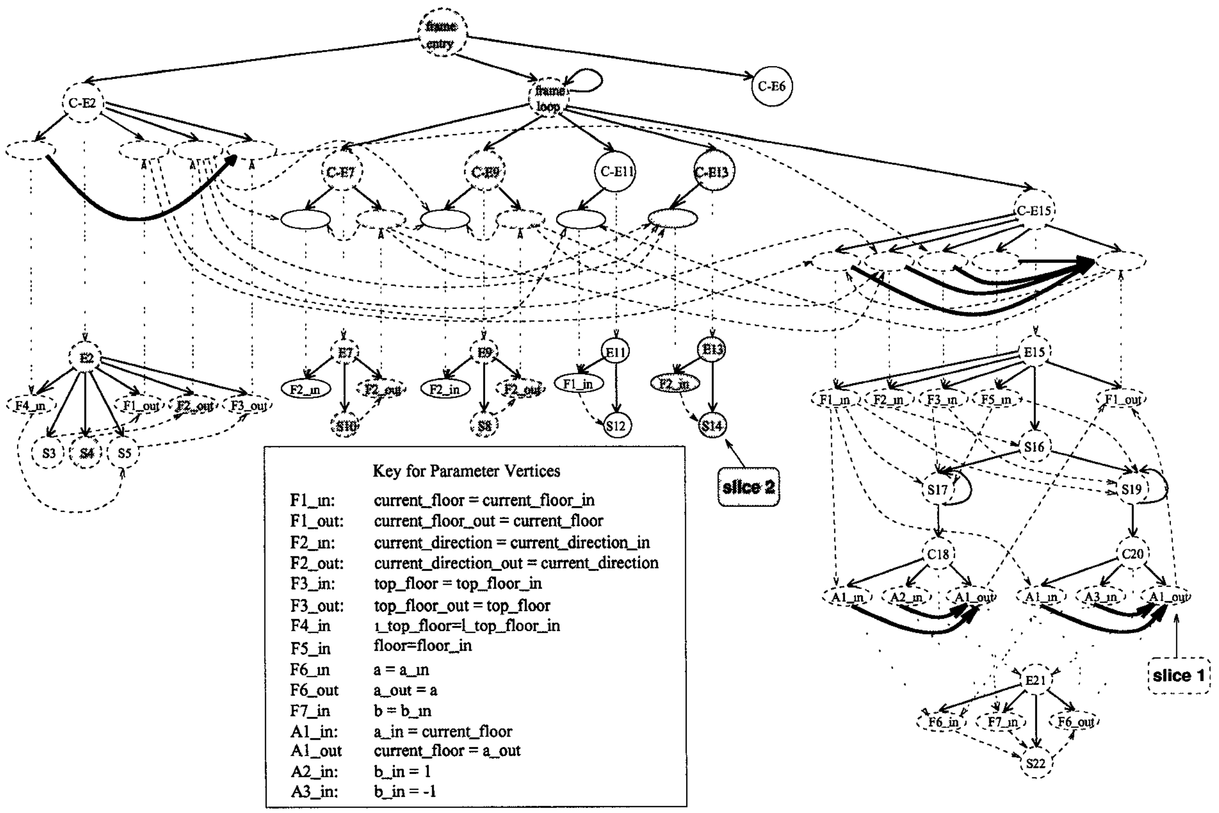
3.2 Incomplete Systems

Classes and class libraries are often developed independently from the applications programs that use them. To represent these incomplete systems for analysis, we simulate possible calling environments using a frame[12]. A *frame* represents a universal driver for a class that lets us simulate the calling of public methods in any order. A frame first calls the constructor of the class and then enters a loop that has calls to each public method in the class. On each iteration through the loop, control can pass to any public method. After the end of the loop, the frame calls the destructor of the class.

A frame is the "main" program of an SDG for an incomplete system. Thus, we use it to construct a procedure dependence graph. The call to the constructor, the frame loop and the call to the destructor are all control dependent on the frame entry vertex; the frame loop is also control dependent on itself. Finally, the frame call is control dependent on the frame loop. The frame call is replaced by calls to the public methods in the class, and parameter vertices are added at the call sites.

To create a procedure dependence graph for a frame for a particular class, we replace the frame call vertex with call vertices for each public method in the class. For example, for the **Elevator** class, shown in Figure 4, the procedure dependence graph for the frame contains C-E2, C-E7, C-E9, C-E11, C-E13, C-E15, and C-E6, which represent call vertices for **Elevator()**, **up()**, **down()**, **which_floor()**, **direction()**, **go()**, and **Elevator()**, respectively. Instead of constructing procedure dependence graphs for each method in the class, we reuse the information in the ClDG. At each call vertex in the frame's procedure dependence graph, we add actual-in and actual-out vertices to match the formal-in and formal-out vertices of the associated method entry vertex in the ClDG for the called method. We connect the ClDG to the procedure dependence graph for the frame by adding parameter-in edges, parameter-out edges, and call edges.

When we connect the procedure dependence graph for the frame to the ClDG for the **Elevator** class, we get the SDG in Figure 4; we omit the class entry vertex and class member edges from the figure. Since there are no source code statements associated with call vertices in the frame, we label them C-Ei, where Ei is the method entry of the called method. We leave actual parameter vertices unlabeled since they just represent copies to and from temporaries. For example, C-E2 is



Key for Parameter Vertices

F1_in:	current_floor = current_floor_in
F1_out:	current_floor_out = current_floor
F2_in:	current_direction = current_direction_in
F2_out:	current_direction_out = current_direction
F3_in:	top_floor = top_floor_in
F3_out:	top_floor_out = top_floor
F4_in	ı_top_floor=l_top_floor_in
F5_in	floor=floor_in
F6_in	a = a_in
F6_out	a_out = a
F7_in	b = b_in
A1_in:	a_in = current_floor
A1_out	current_floor = a_out
A2_in:	b_in = 1
A3_in:	b_in = -1

Figure 4: SDG representation of C^{++} `Elevator` class along with two slices.

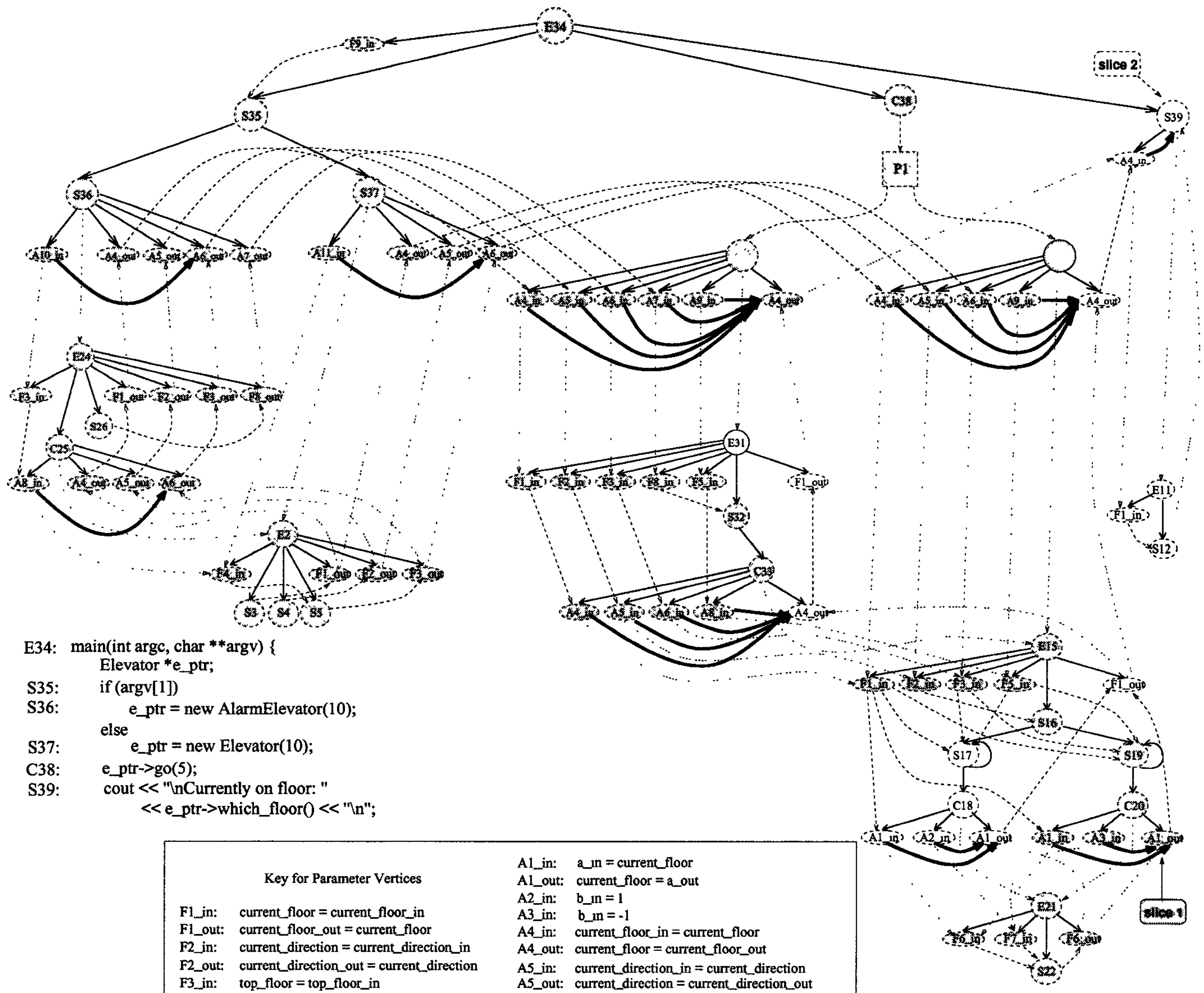
the call vertex associated with method entry E2, and its actual-in and actual-out vertices are unlabeled.

Since instance variables retain their values between method calls, there may be data dependencies across call sites even though the variables are not visible in the calling program. For example, the value of `current_direction` at the end of a call to `up()`, F2_out in Figure 4, can subsequently be used by a call to `go()`. To account for these data dependencies, our construction adds data dependence edges between the actual-out vertices of each method call and the corresponding actual-in vertices of every other method call. For example, the actual-out vertex of C-E7, representing the value of `current_direction`, is connected to every other actual-in vertex corresponding to data member `current_direction`, namely the actual-in vertices for C-E9, C-E13, and C-E15. No values flow into a class constructor except through formal parameters since no previous instance variable values exist; no values flow from a destructor since instance variables are no longer available.

3.3 Complete Programs

We construct the SDG for a complete program by connecting calls in the partial system dependence graph to methods in the ClDG for each class. We described this construction in detail in Section 2; it involves connecting call vertices to method entry vertices, actual-in vertices to formal-in vertices, and formal-out vertices to actual-out vertices. The summary edges for methods in a previously analyzed class are added between the actual-in and actual-out vertices at call sites. This construction of the SDG for an object-oriented system maximizes reuse of previously constructed portions of the representation. The introduction of variables in the scope of the application program, such as a global variable, does not affect the representation in any of the ClDG's. Any global variables referenced or modified by a class must be declared **extern** in the class, so this information would have been included while building the class's ClDG.

Figure 5 contains an example of an application program that instantiates an object. The variable `e_ptr` could point to an object of type `Elevator` or `AlarmElevator`. This graph was constructed by building a partial SDG for the `main` function, including the previously computed representation for the `Elevator` and `AlarmElevator` classes, and connecting each graph using call, parameter-in, and parameter-out edges; we omit class entry vertices and class member edges from the graph. This example also illustrates the way in which the ClDG represents the effects of methods that return values. For example, S39 contains a call to `e_ptr->which_floor` that returns a value (we omit the representation of the call to **cout**). The call vertex is data dependent on the value of `current_floor`. A summary edge is added from S39→A4_in to S39, and a parameter-out edge is added from S12, the return statement, to S39.



```
E34:  main(int argc, char **argv) {
          Elevator *e_ptr;
S35:      if (argv[1])
S36:          e_ptr = new AlarmElevator(10);
          else
S37:          e_ptr = new Elevator(10);
C38:      e_ptr->go(5);
S39:      cout << "\nCurrently on floor: "
               << e_ptr->which_floor() << "\n";
```

Key for Parameter Vertices

F1_in:	current_floor = current_floor_in	A1_in:	a_in = current_floor
F1_out:	current_floor_out = current_floor	A1_out:	current_floor = a_out
F2_in:	current_direction = current_direction_in	A2_in:	b_in = 1
F2_out:	current_direction_out = current_direction	A3_in:	b_in = -1
F3_in:	top_floor = top_floor_in	A4_in:	current_floor_in = current_floor
F3_out:	top_floor_out = top_floor	A4_out:	current_floor = current_floor_out
F4_in:	l_top_floor = l_top_floor_in	A5_in:	current_direction_in = current_direction
F5_in:	floor = floor_in	A5_out:	current_direction = current_direction_out
F6_in:	a = a_in	A6_in:	top_floor_in = top_floor
F6_out:	a_out = a	A6_out:	top_floor = top_floor_out
F7_in:	b = b_in	A7_in:	alarm_on_in = alarm_on
F8_in:	alarm_on = alarm_on_in	A7_out:	alarm_on = alarm_on_out
F8_out:	alarm_on_out = alarm_on	A8_in:	l_top_floor_in = l_top_floor
		A9_in:	floor_in = 5
		A10_in:	top_floor=10
		A11_in:	l_top_floor=10

Figure 5: SDG representation of application program that uses the **Elevator** and **AlarmElevator** classes along with two slices.

4 Slicing

Since the SDGs that we compute for object-oriented programs belong to a class of SDGs defined in Reference [14], we can use the two-pass graph reachability algorithm for computing slices. That slicing algorithm uses a slicing criterion $< p, x >$ in which p is a statement and x is a variable that is defined or used at p. Object-oriented software developers prevent users of a class from directly manipulating instance variables within an object (the object's state) by providing methods as an interface for setting and observing the state of the object. Thus, object-oriented systems substitute many variable references with method calls that simply return a value. To let us slice on the values returned by a method, we use a slightly modified slicing criterion. Our slicing criterion $< p, x >$ consists of a statement p and a variable or a 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. Since our SDG construction creates summary edges that are used to preserve the calling context, we compute slices that satisfy our criterion on our SDGs using the two-pass algorithm. During the first pass of the slicing algorithm, summary edges facilitate slicing across call vertices that have transitive dependencies on actual-in vertices. During the second pass of the algorithm, the algorithm descends into called methods (or procedures) along the parameter-out edges.

We consider slices on incomplete programs, and use the `Elevator` class of Figure 2 for illustration. Figure 4 gives the SDG for the `Elevator` class and shows two slices: slice 1, depicted with dashed line vertices, and slice 2, depicted with shaded vertices. Slice 1 is the backward slice computed at vertex C20→A1_out, which represents the value of `current_floor` returned from the call at C20. The dashed line vertices in Figure 4 indicate this slice, which includes all statements that could affect the value of `current_floor` at C20→A1_out for any sequence of calls to `Elevator`'s public methods. On the first pass, the slicing algorithm marks all dashed line vertices except E21, E21→F6_in, E21→F7_in, E21→F6_out and S22. On the second pass, the algorithm descends into called method `go()`, and marks the rest of the vertices shown with dashed lines in the figure. Vertices that are part of the frame are not included in the slice as their only purpose is to facilitate slicing; however, for completeness, we include them in our illustration.

Since we compute a slice as if all possible sequences of calls to public methods were possible, it includes more statements than would likely be included if the slice were taken from an application program that specified a particular call sequence of public methods. However, during development, this type of slice may be useful since it indicates the dependencies that could exist among statements in the class, and may assist in understanding, debugging or testing the class.

Slice 2 is the backward slice computed at vertex S14, which represents the value of `current_direction` returned by a call to `direction()`. The shaded vertices in Figure 4 indicate this slice, which includes only statements S14, E13, E13→F2_in, S8, E9, E9→F2_out, S10, E7, E7→F2_out, S4, E2, E2→F2_out and the associated frame vertices because these are the only statements that modify the direction of the elevator.

Next, we consider slices on the application program in Figure 5, which gives the SDG for this program and shows two slices: slice 1, depicted with shaded vertices, and slice 2, depicted with dashed line vertices. Slice 1 is a backward slice computed with respect to `current_floor` at C20→A1_out. This application program has a polymorphic call, which causes the slicing algorithm to include statements from all possible destinations of the polymorphic call in the slice.

Slice 2 is a backward slice computed with respect to the call to `which_floor()` at vertex S39, which includes includes all statements that may affect `current_floor`. The summary edge between S39→A4_in and and S39 summarizes the statements upon which the return value of `which_floor()` is dependent. During the first pass of the slicing algorithm, traversal proceeds backward over the data dependence edges that are incident to S39→A4_in, and finds those statements that affect its value. During the second pass, the algorithm traverses back from all vertices marked during the first pass, as well as backward over the parameter-out edge incident on S39. The second pass includes all statements in the `which_floor` method.

A forward slice on a slicing criterion $< p, x >$ includes all statements affected by the value of x at p. Horwitz, Reps, and Binkley[14] also describe a two-phase algorithm for computing forward slices on a system dependence graph. We can apply their forward slicing algorithm to our system dependence graphs to calculate forward slices.

5 System Dependence Graph Size

We designed our SDGs for object-oriented software so that existing slicing algorithms could be applied to them. Thus, our algorithm for constructing SDGs for object-oriented programs is similar to existing algorithms for procedural language programs except that it reuses partial SDGs, such as those constructed for individual classes, whenever possible. However, since our SDGs are constructed for object-oriented software, there may be differences in their sizes compared to SDGs for procedural language programs. In this section we discuss the size of our SDGs.

Table 1 lists the variables that contribute to the size of an SDG. We give a bound on *ParamVertices*, and use this bound to compute the upper bound on the size of a method or procedure.

$$\begin{aligned} ParamVertices(m) &= Params \\ &\quad + Globals \\ &\quad + InstanceVars \qquad (1) \\ Size(m) &= O(Vertices + CallSites \\ &\quad * (1 + TreeDepth \\ &\quad * (2 * ParamVertices(m))) \\ &\quad + 2 * ParamVertices(m)) \qquad (2) \end{aligned}$$

Given *Methods*, the number of methods in the entire system, the upper bound on the number of vertices in an SDG, including all classes, is:

$$Size(SDG) = O(Size(m) * Methods) \qquad (3)$$

Size(*SDG*) is a rough upper bound on the number of vertices in an SDG that we construct. In practice, an SDG may be considerably more space efficient for several reasons. First, the computation of GMOD/GREF sets[14] can considerably reduce the number of global variables and instance variables that must be represented as parameter vertices at call and entry sites for a particular method.

Second, the computation of *Size*(*SDG*) assumes that all method calls are indirect calls. C^{++} programs

Table 1: Parameters affecting the size of an SDG

Vertices	Greatest number of predicates and assignments in a single method or procedure
Edges	Greatest number of edges in a single method or procedure
Params	Greatest number of formal parameters in any method or procedure
Globals	Number of global variables in the system
InstanceVars	Greatest number of instance variables in a class, including those in all instantiated classes
CallSites	Greatest number of call sites in any method or procedure
TreeDepth	Depth of inheritance tree determining number of possible indirect call destinations
Methods	Number of methods or procedures in the system

tend to use method calls more frequently than C programs, and C^{++} programs often use method calls and objects as parameters to methods. Despite this, the full cost of a method call is only incurred for calls to virtual methods whose destination cannot be resolved statically. Calder and Grunwald[7] report that 80% of the function calls in C^{++} are calls to methods and that only 67% of these calls are indirect. Furthermore, using execution profiles of C^{++} programs, they determined that the target of most indirect function calls can be accurately predicted. Recently, Pande and Ryder[22] presented an analysis technique that statically determines the target of indirect function calls for C^{++} programs. Their experiments corroborate the results of Calder and Grunwald. Static analysis can greatly reduce the number of call sites associated with a polymorphic choice vertex in our SDGs.

Another property of object-oriented programs also helps minimize the construction of an SDG. C programs often use global variables indiscriminately, whereas C^{++} programs encapsulate "global" variables into classes where they are only visible to classes actually requiring access. These factors suggest that although the upper bound on the size of the graph could result in very large SDGs, in practice we expect them to be much smaller.

To demonstrate the actual sizes of SDGs that we expect, we performed a case study on a C^{++} program containing nine classes, 65 methods, and class hierarchies up to three classes deep. Our goal was to compare the number of vertices at call sites and method entries in the computed size and the actual size. We first computed the upper bound for the sample program using Equation 3, which resulted in an SDG with 282,112 vertices. Then, we computed the SDG for the sample program without considering *Vertices*, and it contains only 1,257 vertices. Although our case study is not conclusive, we believe that it is indicative of the sizes of SDGs for object-oriented systems. We are currently developing tools to construct SDGs for C^{++} systems, so that we can perform experiments.

We performed our analysis and bounds computation for C^{++} systems. Since object-oriented programming languages differ, the size of an SDG may vary depending on the language being represented. However, our results are applicable for other statically-typed object-oriented programming languages such as Ada-95.

6 Conclusions

We have presented system dependence graphs for object-oriented software on which efficient interprocedural slicing can be performed. Each system dependence graph consists of a program dependence graph, which represents either the "main" program in the system or a simulation of a calling environment, and class dependence graphs, which represent classes in the system. Class dependence graphs are constructed for each class in the system, and reused in constructing other class dependence graphs or system dependence graphs. Our class dependence graphs are efficiently constructed for derived classes and interacting classes by incorporating parts of previously constructed class dependence graphs. We represent a calling environment for an incomplete system using a frame that simulates all possible calling environments. A frame allows slices to be calculated not only on complete object-oriented applications, but also on individual classes.

We described the computation of slices on our system dependence graphs using an efficient two-pass algorithm. Although our discussion focused on backward slicing, our techniques are also applicable for the calculation of forward slices. Slicing object-oriented programs is relatively efficient because most applications reuse components and it is possible to analyze a component once and reuse the analysis information many times. We are currently implementing our techniques for C^{++}, to experiment with useful applications of slicing such as performing optimizations and gathering metrics.

Acknowledgements

Jo Anna Madril edited many of the figures. Gregg Rothermel, Devidas Gupta, Maria Demetriou, and the anonymous reviewers provided suggestions that improved the presentation of the paper.

References

[1] O. Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 91–107, October 1995.

[2] 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.

[3] H. Agrawal, R. DeMillo, and E. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 60–73, 1991.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[5] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[6] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of Conference on Software Maintenance*, pages 41–50, November 1992.

[7] B. Calder and D. Grunwald. Reducing indirect function call overhead in C^{++} programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Progamming Languages*, pages 397–408, January 1994.

[8] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, 1993.

[9] J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[10] 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.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[12] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 154–163, December 1994.

[13] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[14] 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.

[15] D. Jackson and E. J. Rollins. A new model of program dependence for reverse engineering. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering*, pages 2–10, December 1994.

[16] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[17] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[18] P. E. Lividas and S. Croll. Static program slicing. Technical Report SERC-55F, University of Florida, Software Engineering Research Center, Computer and Information Sciences Department, January 1992.

[19] B. A. Malloy, J. D. McGregor, A. Krishnaswamy, and M. Medikonda. An extensible program representation for object-oriented software. *ACM Sigplan Notices*, 29(12):38–47, December 1994.

[20] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, April 1984.

[21] H. Pande and B. G. Ryder. Static type determination in C^{++}. In *Proceedings of the Sixth USENIX C^{++} Technical Conference*, pages 85–97, April 1994.

[22] H. D. Pande and B. G. Ryder. Static type determination and aliasing for C^{++}. Technical Report LCSR-TR-250, Rutgers Univiversity, July 1995.

[23] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of Second ACM Conference on Foundations of Software Engineering*, pages 11–20, December 1994.

[24] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of Conference on Software Maintenance*, pages 14–25, September 1994.

[25] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.