A Coherent Family of Code-Based Graph Representations for Object-Oriented Software

Mary Jean Harrold
Computer and Information Science
Ohio State University
395 Dreese Lab
Columbus, OH 43210
harrold@cis.ohio-state.edu

Gregg Rothermel
Computer Science
Oregon State University
307A Dearborn Hall
Corvallis, OR 97331
grother@cs.orst.edu

Abstract

Many software engineering techniques rely on code-based graph representations of software, such as control flow graphs or program dependence graphs. To apply these techniques to object-oriented software, we cannot necessarily use existing graph representations of procedural software. Graphs for object-oriented software, like those for procedural software, must represent individual procedures (methods) and entire programs; however, they must also represent classes and their interactions, and account for the effects of inheritance, scoping, persistence, polymorphism, and dynamic binding. These representations should facilitate the use of existing program-analysis techniques, rather than require creation of new techniques. A system for constructing and managing these representations should take advantage of the code reuse inherent in object-oriented software, by reusing representational components. This paper describes a coherent family of code-based graph representations of object-oriented software that meets the foregoing requirements, and outlines the architecture of an extensible system for constructing and managing those representations.

1 Introduction

Many software engineering tools and techniques require program-analysis information. For example, techniques for data-flow testing require data-flow information, and techniques for debugging, regression testing, and impact analysis use program slices. These techniques typically use code-based graph representations such as control flow graphs and control-dependence graphs.

To apply these techniques to object-oriented software, we must first construct appropriate representations for that software. Code-based representations for procedural software are well established, and techniques for constructing these representations are well known. To establish similar representations for object-oriented software, however, we must account for language characteristics that, by uniqueness or frequency of use, distinguish object-oriented software from procedural software. Representations for object-oriented software must efficiently and effectively model classes, objects, inheritance, scoping, persistence, polymorphism, and dynamic binding. Furthermore, because classes are often developed, maintained, and tested independently of calling environments, class representations for both base classes and derived classes should be independently analyzable.

Many existing graph representations for object-oriented software, such as the object, dynamic, and functional views that constitute the Object Modeling Technique (OMT) [16], are not constructed from the code. Thus, these representations do not support code-level applications of techniques such as data-flow analysis and slicing. Other representations are constructed from the code but fail to meet some of the requirements outlined above. The code-based techniques described in [8, 9, 10] omit control dependence information, and

do not provide analysis information about interacting methods. Thus, these techniques do not facilitate tasks, such as slicing across methods in classes, that require that information. The graph defined in [14] represents control flow, data flow and control dependence. However, that representation is not defined for incomplete programs, and thus, does not accommodate independent analysis of classes and objects. Furthermore, in an effort to reduce complexity, the representation has been modified from its procedural counterpart such that it does not support the use of existing algorithms for data-flow analysis and slicing. We discuss these techniques further in Section 5.

To facilitate code-based analyses, we have developed a coherent family of code-based graph representations for object-oriented software that meet the requirements outlined above. Our representations have several important features.

- The representations account for characteristics of object-oriented software: they model classes and objects, inheritance hierarchies, and calling relationships, and capture control flow, control dependence, and data dependence. By modeling these features and relationships, the representations facilitate the use of data-flow analysis and slicing techniques, which in turn support the use of program-analysis-based techniques for regression test selection, data-flow testing, and impact analysis. However, because we have designed the representations by careful extension of existing representations for procedural language software, our representations support the application — directly or with minor modifications — of existing program analysis algorithms.
- The representations incorporate a methodology that simulates arbitrary sequences of calls to public methods in a class. This methodology facilitates independent analysis of classes and interacting classes. In this case, also, our representations support the application — directly or with minor modifications — of existing program analysis algorithms.
- The design of our representations facilitates the reuse of representational components within our family of representations. Also, our system for managing these representations takes advantage of the code reuse inherent in object-oriented software, to provide additional reuse of representational components. This reuse, and the architecture of our system, promote system extensibility: this facilitates extensions to support new representations and new analysis techniques.

This paper presents our family of graph representations, discusses some applications of our representations, and outlines the architecture of our system for building and managing these representations.

2 A Family of Graphical Representations

This section describes our family of code-based representations for object-oriented software. We first describe the class hierarchy graph, which can be constructed using a class's definition. Then, we present the class call graph, the class control flow graph, and the class dependence graph, which are constructed using class implementations. Finally, we discuss framed graphs, which can be used for various types of class analyses. For each representation, we provide background information required to understand that representation. Then, we discuss the use of the representation at the *class level*: for single classes or classes that are related by inheritance. Finally, we show how to extend the representation to the *interclass level*: for interacting

```
class Coinbox {
public:
  Coinbox::Coinbox() {
    totCoins=curcoins=alVend=0;
    checkflag = 0;
  }
  Coinbox::Coinbox(unsigned flag) {
    totCoins=curCoins=alVend=0;
    checkflag = flag;
  }
  Coinbox::~Coinbox(){}
  virtual void Coinbox::insert() {
    curCoins++;
    if (curCoins>1) alVend=1;
    if (checkflag) check();
  }
  void Coinbox::vend(unsigned selection) {
    if (alVend) {
      totCoins+=curCoins;
      curCoins = alVend = 0;
      d1.dispense(selection);
    }
  }
  void Coinbox::retrn() {
    c1.retcoins(curCoins);
    curCoins = 0;
    if (checkflag) check();
  }
protected:
  unsigned totCoins, curCoins, alVend;
  unsigned checkflag;
  Dispenser d1;
  Coinreturn c1;

  void Coinbox::check() {
    if ((curCoins<2)&&(alVend)) {
      cout<<"Invariant violated:";
      cout<<"User can vend
              for free!" << endl;
    }
  }
};
```

```
class Dispenser {

public:

  Dispenser::Dispenser() {}

  Dispenser::~Dispenser() {}

  void Dispenser::dispense(unsigned choice) {
    cout<<"User:  Take drink number " <<
           selection << endl;
  }

};
```

```
class Coinreturn {

public:

  Coinreturn::Coinreturn() {}

  Coinreturn::~Coinreturn() {}

  void Coinreturn::retcoins(unsigned number) {
    cout<<"User:  Take your " << number
           " coins" << endl;
  }
};
```

Figure 1: C++ code for `Coinbox` (top), `Dispenser` (lower left), and `Coinreturn` (lower right).

classes in different class hierarchies. For representations at the class level, calls to methods in classes that are not in the hierarchy are not expanded, whereas in the interclass level representations, such calls are expanded.[1]

We use a vending machine example[2] throughout our presentation. Figure 1 gives the C++ code for the coinbox component, `Coinbox`, of the vending machine, along with the other components with which it interacts. For brevity, we omit some implementation details, such as `#include` directives, from the figure. When a user inserts a coin into the vending machine, `insert` increments a count of coins, `curCoins`. After the user has inserted at least two coins in the vending machine, `insert` sets a flag, `alVend`. When a user attempts to make a selection, `vend` checks the current value of `alVend`. If `alVend` is set, `vend` adds `curCoins` to the total number of coins, `totCoins`, in the coinbox, resets the appropriate state variables, and sends a message to a drink dispenser, `Dispenser`, which dispenses the user's selection. In the code listed, the

[1] In practice, the interclass level might selectively expand calls.

[2] This example is adapted from one originally presented by Kung, et al.[11].

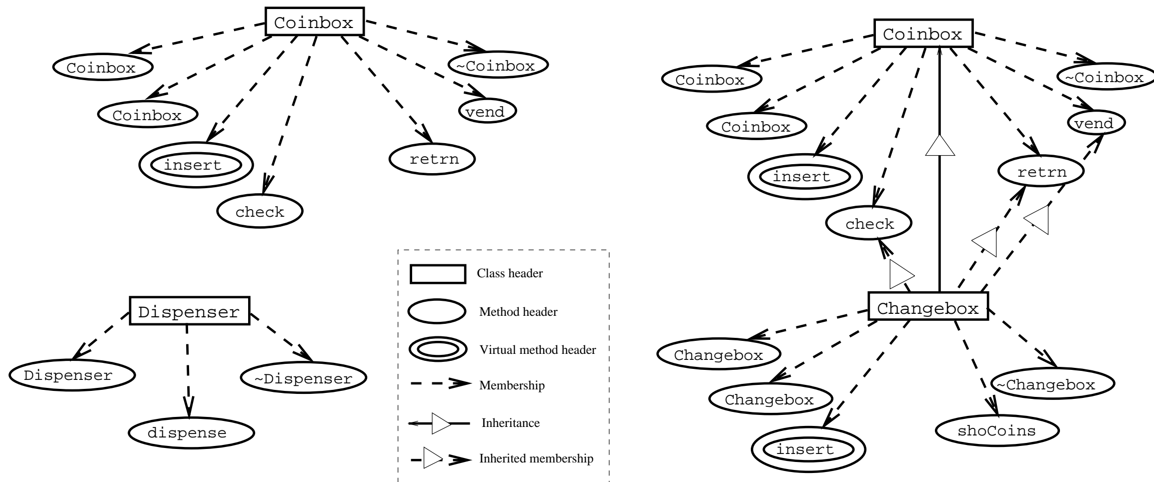

Figure 2: Class hierarchy graph for `Coinbox` (upper left), `Dispenser` (lower left), and `Changebox` (right).

`dispense` method simulates dispensing of a drink by outputting a message. At any time, the user may request a return of coins, in which case `retrn` sends a message to a coin returner, `Coinreturn`, and resets `curCoins`; in the code listed, the `retcoins` method simulates returning of coins by outputting a message. In addition to public methods `insert`, `vend`, and `retrn` that provide operations on the coinbox, `Coinbox` contains private method `check` that reports violations to the class invariant; both `insert` and `retrn` call `check`. `Coinbox` has one constructor that creates a `Coinbox` without this checking code, and one constructor that creates a `Coinbox` with this checking code.[3] Finally, `Coinbox` has one destructor, `~Coinbox`.

2.1 Class Hierarchy Graphs

A *class hierarchy graph* represents a class inheritance hierarchy, and the methods that participate in the hierarchy. A class hierarchy graph represents the definition view of the software but not the implementation details of individual methods. A class hierarchy graph contains a single *class header* node, and contains a *method header* node for each method in the class. The class header node is connected to method header nodes by *membership* edges. For a single class, a class hierarchy graph simply represents membership in the class. For example, Figure 2 depicts the class hierarchy graphs for the `Coinbox` and `Dispenser` classes (upper left and lower left, respectively).

For a derived class, a class hierarchy graph represents the relationships between the class and the classes from which it inherits, along with the new methods defined in the class. In addition to membership edges, a class hierarchy graph for a derived class contains *inheritance* edges, which connect the class header node for the derived class to the class header nodes for its superclasses, and *inherited membership* edges, that connect the class header node for the derived class to the method header nodes of methods that it inherits. For example, consider class `Changebox`, whose code is shown in Figure 3. `Changebox` is derived from `Coinbox` of Figure 1. Whereas a `Coinbox` returns no coins to a user who has inserted more than two coins before making a selection, a `Changebox` returns any extra coins that the user inserts. A `Changebox` also manages a display that indicates the number of coins that the user has inserted; in the code listed, the `Display::coinsinwindow` method simulates the display of the number of coins by outputting a message. Figure 2 shows the class

[3] We could also implement this checking functionality using a single constructor; we use two simply for purposes of illustration.

```
class Changebox : public Coinbox {

public:

  Coinbox::Changebox::Changebox() :
      Coinbox() {}

  Coinbox::Changebox(unsigned flag) :
      Coinbox(flag) {}

  Coinbox::~Changebox() {}

  void Changebox::insert() {
    if  (curCoins > 1)
      c1.returncoins(1);
    else
      Coinbox::insert();
  }
  void Changebox::shocoins() {
    d3.coinsinwindow(cur);
  }
protected:
  Display d3;
};
```

```
class Display {

public:
  Display::Display() {}

  Display::~Display() {}

  void Display::
      coinswindow(unsigned number) {
   cout << "User:  You have inserted "
   number << " coins" << endl;
  }
};
```

Figure 3: C++ code for `Changebox`, a subclass of `Coinbox` (left), and `Display` (right).

hierarchy graph for the `Changebox` class (right). In the figure, an inheritance edge connects `Changebox`'s class header node with `Coinbox`'s class header node. The `Changebox` constructors and destructor, along with methods `insert` and `shoCoins`, are new in `Changebox` and thus, connected to `Changebox`'s class header node by membership edges. Methods `retrn`, `vend`, and `check` are inherited from `Coinbox` and thus, connected to `Changebox`'s class header node by inherited membership edges.

Class hierarchy graphs are not defined at the interclass level.

A class hierarchy graph has various uses. For example, one class testing technique[4] exploits the hierarchical nature of the inheritance relation to test a group of related classes. This technique can use the class hierarchy graph to determine which tests for a base class can be reused to test a derived class. The class hierarchy graph also provides information that can be useful for computing metrics for object-oriented software[2, 17], such as numbers of data attributes, numbers of methods, visibility information, numbers of base classes, and depth and width of inheritance hierarchies.

2.2 Class Call Graphs

A *call graph* represents static calling relationships among procedures. Nodes in a call graph represent procedures. An edge from node P_1 to node P_2 represents the fact that procedure P_1 calls procedure P_2.

A *class call graph* represents calling relationships among methods in a class hierarchy. Both the class call graph and the class hierarchy graph contain class header nodes, method header nodes, virtual method header nodes, membership edges, inheritance edges, and inherited membership edges. A class call graph, however, also includes edges that represent method calls.

For a single class, a class call graph represents caller/callee relationships between methods that are members of that class, but does not account for invocations of methods that are members of other classes. When method M_1 in class C calls method M_2, if M_2 is also in C, the class call graph for C contains a *call*

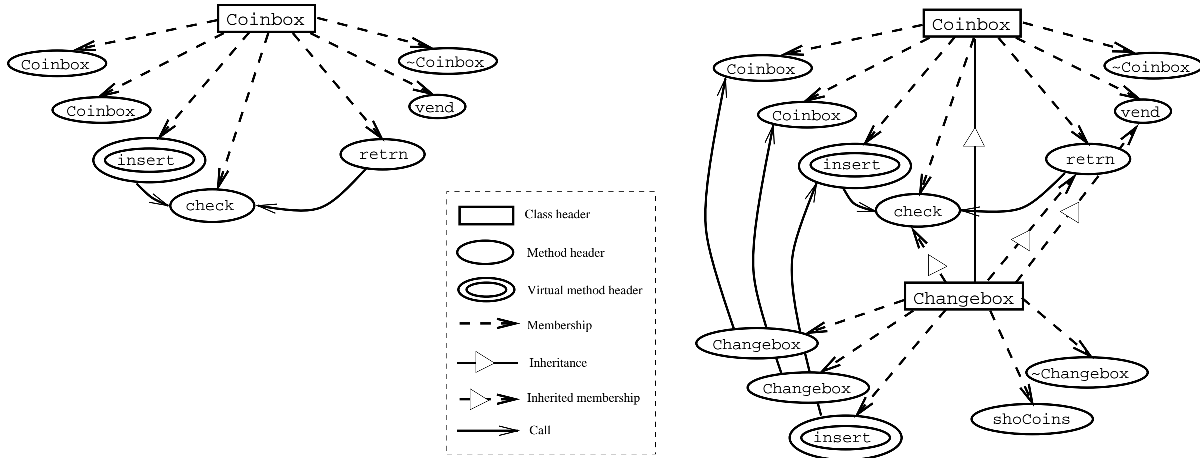

Figure 4: Class call graphs for `Coinbox` (left), and `Changebox` (right).

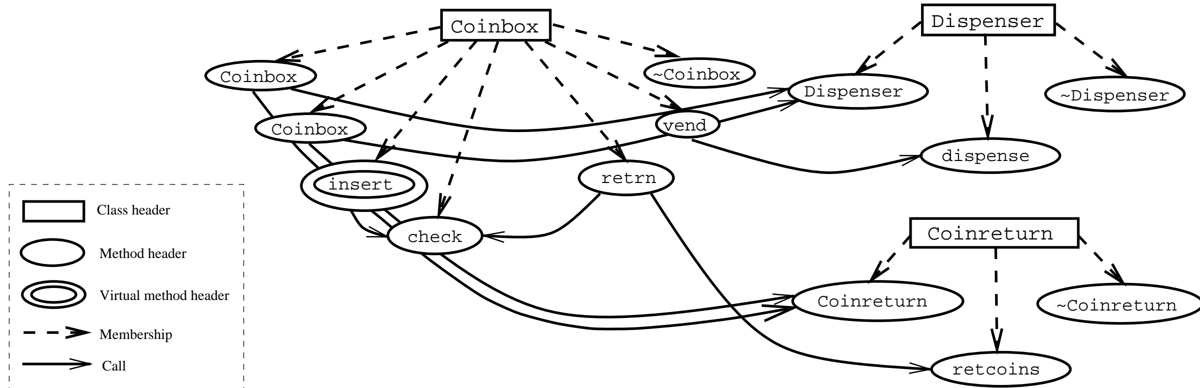

Figure 5: Interclass call graph for `Coinbox`.

edge that connects the method header of M_1 to the method header of M_2. Figure 4 depicts the class call graph for `Coinbox` (left). Both `insert` and `retrn` call method `check`; thus, the class call graph for `Coinbox` contains call edges (`insert`, `check`) and (`retrn`, `check`).

Inheritance creates interactions between classes. Class call graphs for derived classes represent such interactions. For example, Figure 4 shows the class call graph for `Changebox` (right). The class call graph for `Changebox` is constructed from the class call graph of its parent class, `Coinbox`, with additional nodes and edges that represent the derived class. In `Changebox`, both constructors call constructors in `Coinbox`, and virtual method `insert` calls virtual method `insert` in `Coinbox`. Call edges in the class call graph represent these interactions.

When methods in one class send messages indirectly or directly to methods in other classes, these messages represent interclass interactions. We refer to a class call graph that represents such interclass interactions as an *interclass call graph*. Methods may indirectly send messages to methods in other classes by using object declarations. For example, the instantiation of object `d1` of type `Dispenser` in `Coinbox` causes the constructor `Dispenser` to be executed. Because this call actually occurs when a `Coinbox` is instantiated, we associate the call with the `Coinbox` constructor. Thus, for every object declared in a class, there is a

call edge from each of the class's constructors to the constructor of the instantiated object's class. Figure 5 shows the interclass call graph for `Coinbox`. Because `Coinbox` declares object `d1` of type `Dispenser`, the graph contains call edge (`Coinbox`, `Dispenser`) from each `Coinbox` constructor; similarly, the graph contains call edge (`Coinbox`, `Coinreturn`) from each `Coinbox` constructor.

Methods may also directly send messages to methods in other classes. For example, after the instantiation of object `c1` in class `Coinbox`, method `retrn` sends a message to method `retcoins` to return `curCoins` to the user. Call edge (`retrn`, `retcoins`) in the interclass call graph of Figure 5 represents this message.

Class call graphs can be used for several types of analysis. Analysis techniques that use the graphs make worst-case assumptions about called methods, unless additional information is known. For example, a flow-insensitive data flow analysis technique assumes that any variable in scope in a called method may be defined and used during any call to the method. Call graphs also play a role in the construction of other interclass representations, as we shall show.

2.3 Class Control Flow Graphs

Control flow graphs have long been used to represent procedures in procedural-language programs. We can also use control flow graphs to represent individual class methods. A *control flow graph*[1] for method M contains a node for each statement in M; edges between nodes represent flow of control between statements. Labeled edges leaving nodes associated with conditional statements represent control paths taken when the condition evaluates to the value of the edge label. The method header node represents the unique entry to M, and a unique exit node represents exit from M.

A *class control flow graph* represents the static control flow relationships that exist within and among methods of that class. A class control flow graph consists of a class call graph, in which each method header node is replaced by the control flow graph for its associated method. Furthermore, when a call node represents an invocation of a method that is a member of a class in the hierarchy, and thus, whose control flow graph is available as part of the class control flow graph, we split the call node into *call* and *return* nodes. We use a call edge to connect the call node to the method header node of the called method. We also insert a return edge from the exit node of the control flow graph for called method to the return node of the graph for the calling method.[4] If the called method is not in the hierarchy, we postpone splitting the call node into call and return nodes until we want to consider interclass interactions.[5]

Figure 6 shows a partial class control flow graph for the `Coinbox` class; we omit the control flow graphs for one `Coinbox` constructor, and for the `Coinbox` destructor, and shade the method headers for these nodes to indicate that they are unexpanded. Notice the expansion of call sites depicted in the figure. For example, the call to `check` in `insert` is represented by both a call node and a return node; a call edge connects the call node in `insert` to the method header node for `check`, and connects the exit node for `check` to the return node in `insert`.

A class control flow graph also represents interactions that arise due to inheritance. For a derived class, a class control flow graph inherits nodes and edges from the class control flow graph for the parent class; the

[4] In this sense, our graphs resemble the *interprocedural control flow graphs* defined originally by Landi and Ryder[12].

[5] In Reference [5], we defined class control flow graphs to include a *frame*, which we discuss in Section 2.5 of this paper. We have since discovered that a more general approach defines the class control flow graph without the frame because this graph can be used either with a frame for analysis of classes or interacting classes, or with representations of calling programs for analysis of those programs.

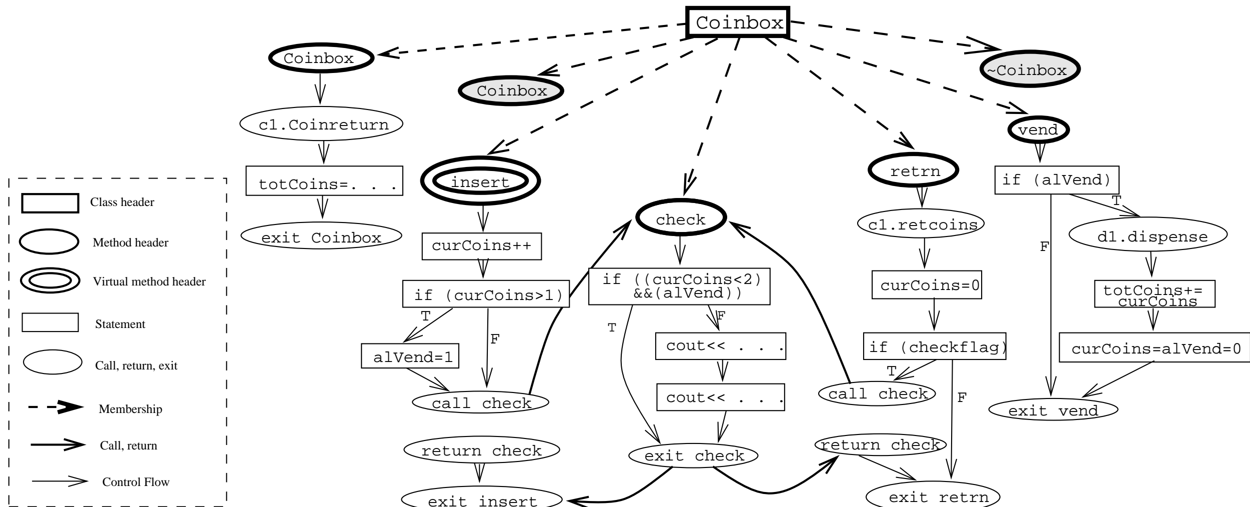

Figure 6: Class control flow graph for `Coinbox`; shaded nodes represent unexpanded portions of the graph.

graph also includes nodes and edges that represent control flow for methods newly defined for the class.

When classes interact due to declarations of objects, or calls to methods defined in other classes, we connect the class control flow graphs of the individual classes to form an *interclass control flow graph*. Classes that interact because of calls to virtual methods whose destination is dynamically bound, however, require special handling. For example, consider class `Vending`, whose C++ code is given in Figure 7. Class `Vending` contains a constructor, a destructor, and a method, `insert`, that calls either the `insert` method in `Coinbox` or the `insert` method in `Changebox`. The `Vending` constructor accepts a parameter, `which`, that indicates which of the coinbox classes will be used in a particular instantiation of a `Vending` object. The statement "`cptr -> insert();`" in method `Vending::insert()` is a polymorphic call. If `which` is 0, the statement references an object in `Coinbox`; otherwise, the statement references an object in `Changebox`.

To accommodate polymorphic calls, we introduce *polymorphic call* and *polymorphic return* nodes. For each possible destination at a call statement that represents a polymorphic call, we add a polymorphic call node and a polymorphic return node. A call edge connects the call statement to each polymorphic call node, and a call edge connects each polymorphic call node to the associated destination method header. A return edge connects the exit node of a destination method with the associated polymorphic return node, and a return edge connects each polymorphic return node with the single return node associated with the call statement.

To illustrate our representation of polymorphic calls, consider the partial interclass control flow graph for `Vending`, shown in Figure 8; the figure shows only those parts of the class control flow graphs for `Coinbox` and `Changebox` that interact with class `Vending`. Because an object of type `Vending` may contain either a `Coinbox` or a `Changebox`, the constructor for an object of type `Vending` contains a call to a constructor for `Coinbox` and a call to a constructor for `Changebox`. Associated with the polymorphic call `cptr -> insert()` in method `insert` in `Vending` are polymorphic call and polymorphic return nodes, shown in the figure as dotted ellipses. Call edges connect the two polymorphic call nodes to method header nodes for `Coinbox::insert` and `Changebox::insert`, respectively. Similar return edges connect the method exit nodes to the polymorphic return nodes.

```
class Vending {

public:

  Vending::Vending(unsigned which) {
     if (which==0)
        cptr = new Coinbox;
     else
        cptr = new Changebox;
    }
  Vending::~Vending() {}

  Vending::insert() {
      cptr -> insert();
   }
protected:
   Coinbox *cptr;
};
```

Figure 7: C++ code for `Vending` class.

Class control flow graphs directly support the use of flow-sensitive data flow and alias algorithms[12]. The graphs thus facilitate data flow testing of interacting methods[6]. Class control flow graphs can also be used to gather metrics about the methods and interacting methods in the class.

2.4 Class Dependence Graphs

Class dependence graphs are based on the system dependence graphs initially defined by Horwitz, Reps, and Binkley[7]. A *system dependence graph* represents control dependencies and data dependencies for an entire program by connecting program dependence graphs for each procedure in the program. A *program dependence graph*[6] for procedure P consists of the nodes in the control flow graph for P, and two types of edges: *control dependence edges*, which represent control dependencies in P, and *data dependence edges*, which represent data dependencies in P. For nodes X and Y in a control flow graph, if Y is *control dependent* on X then there are two paths out of X, such that one path necessarily reaches Y, and the other path may not reach Y. We say that Y is control dependent on X with the label 'T' (true) or 'F' (false). For nodes X and Y in a control flow graph, if Y is data dependent on X, X defines a variable V that is used at Y.[7]

A class dependence graph represents the control and data dependence relationships among statements in the methods within a single class hierarchy. Like a system dependence graph, a class dependence graph connects individual program dependence graphs for methods that are members of the class.

Figure 9 depicts a partial class dependence graph, at the class level, for the `Coinbox` class. Individual program dependence graphs contain all types of nodes found in control flow graphs: namely, *statement*, *call*, *exit*, and *method header* nodes. The graphs also contain *parameter* nodes that were not present in the class control flow graph: these nodes model parameter passing. At each entry node, there is a *formal-in* parameter node for each formal parameter, and a *formal-out* parameter node for each formal parameter that may be modified by the method. At each call node, there is an *actual-in* parameter node for each actual parameter, and an *actual-out* parameter node for each parameter that may be modified by the method. Global variables

[6]Although called a "program dependence graph" for historical reasons, these graphs actually represent dependencies for single procedures.

[7]Although there are other types of data dependence, we consider only *flow dependence.*

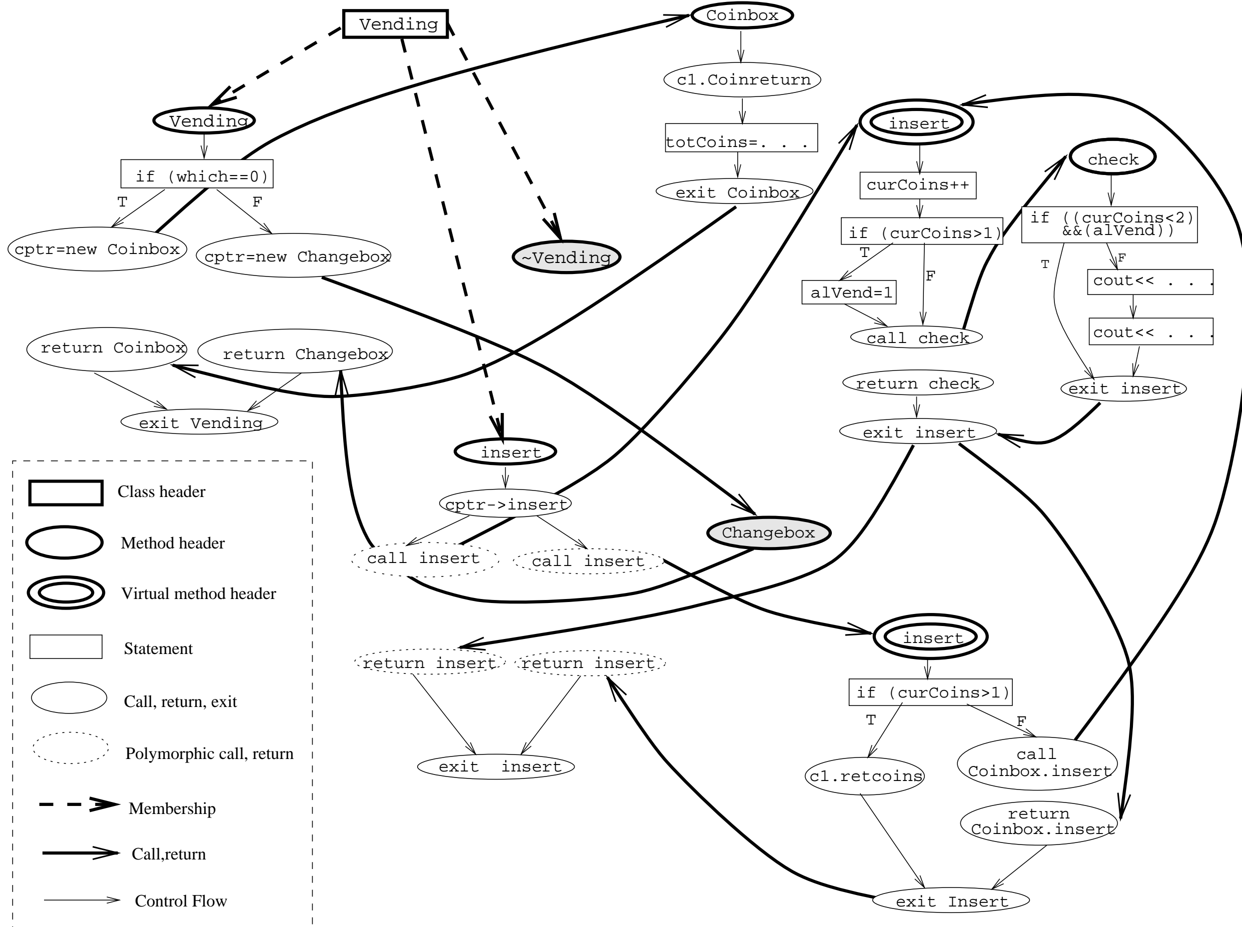

Figure 8: Partial interclass control flow graph for `Coinbox`; shaded nodes represent unexpanded portions of the graph.

---

are treated like parameters at call and method header sites. In the figure, we depict actual-in and formal-in nodes to the left of their associated call or entry node, and we depict actual-out and formal-out nodes to the right of their associated call or method header nodes.

Class dependence graphs contain several types of edges. Like call edges in the class control flow graph, call edges in class dependence graphs connect call nodes to the method header nodes of methods that they invoke. *Control dependence* edges represent dependencies that exist between nodes (statements) within particular methods. *Data dependence* edges represent data dependencies between statements within particular methods. (For simplicity, we omit data dependence edges from our figures.) *Parameter* edges connect actual-in parameter nodes to corresponding formal-in parameter nodes, and connect actual-out parameter nodes to formal-out parameter nodes. *Summary* edges, which connect actual-in parameter nodes to actual-out parameter nodes, model the transitive flow of dependence across method calls. Note that at calls to methods that are not members of the class hierarchy, we do not attach parameter nodes.

Interclass dependence graphs represent the control and data dependencies for interacting class that are not in the same hierarchy. For calls to methods in other hierarchies, we use a technique that is similar to the one used for class dependence graphs, and add parameter nodes and parameter edges. For polymorphic

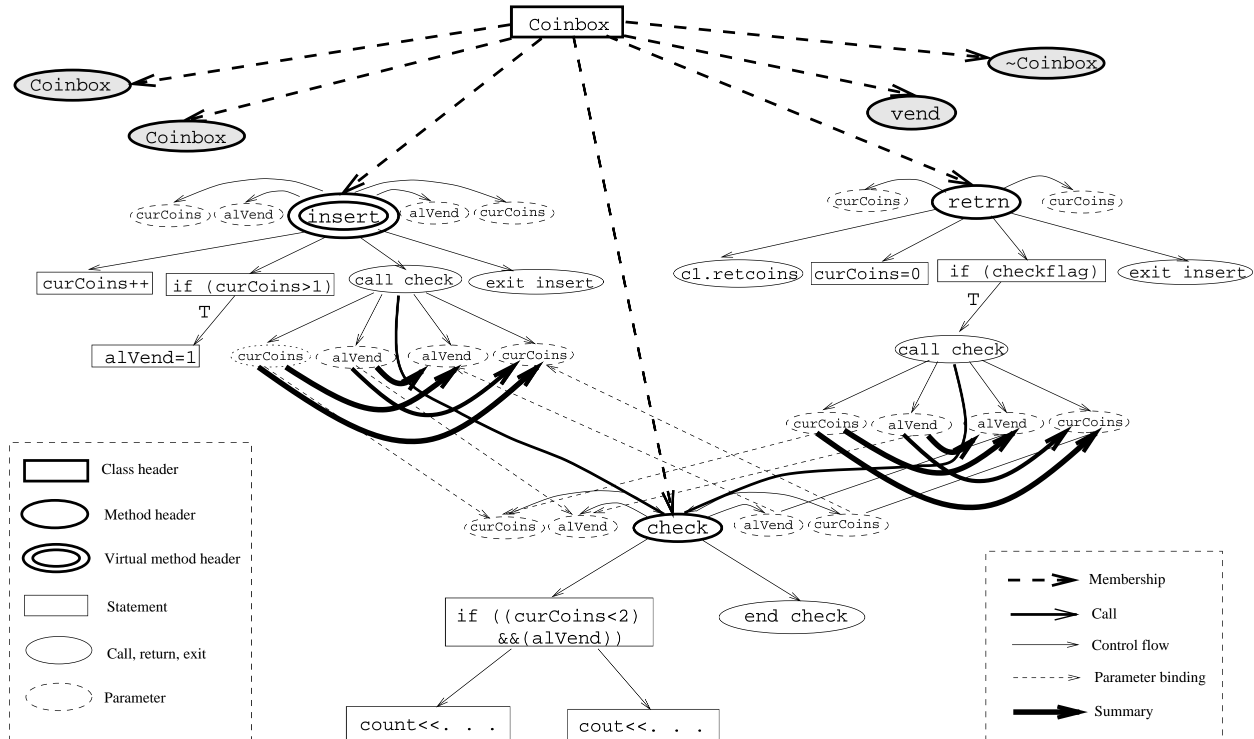

Figure 9: Partial class dependence graph for `Coinbox`; shaded nodes represent unexpanded portions of the graph.

calls, we use a technique that is analogous to the one used in the construction of the interclass control flow graph, adding polymorphic call nodes. However, we also add formal-in and formal-out parameter nodes that match the actual-in and actual-out parameter nodes at the method entry node of the called method.

Class dependence graphs directly support the use of forward and backward slicing techniques, originally defined by Horwitz, Reps, and Binkley[7], that can be applied to the interacting methods in the class. Also, the regression testing technique of Reference [15] uses a similar graph representation.

2.5 Framed Graphs

In the previous subsections, we described various representations for classes and interacting classes. Many of the analysis techniques that we want to perform on these classes function only on complete program representations. When an applications program uses a class, we can construct representations for the applications program, reusing portions of class graphs or interclass graphs where the program uses classes, and analyze those representations. However, because classes are developed independent of calling environments, we may also want to perform analysis on individual classes or interacting classes independent of such environments. This section presents a technique that enables analyses of such incomplete programs. We first discuss the overall technique, and then discuss ways in which we use it to create class representations that can be used for analysis.

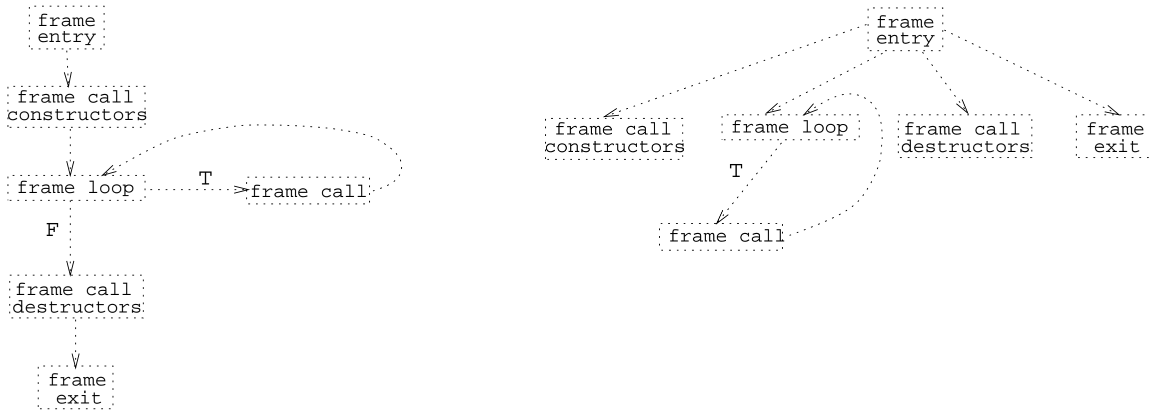

Figure 10: Unexpanded control flow and control dependence graphs for a frame.

The Frame

To represent a class in a manner that supports the use of existing program analysis tools, we find it useful to add a frame to the class. A *frame* is an abstraction of a driver program that simulates arbitrary sequences of calls to public class methods. A frame can be attached to class call graphs, class control flow graphs, or class dependence graphs, yielding *framed class call graphs*, *framed class control flow graphs*, and *framed class dependence graphs*, respectively. Frames can also be attached to the interclass versions of these graphs. In the next two subsections, we discuss two of these framed graphs.

Figure 10 shows the control flow and program dependence graphs for a frame. A frame consists of six vertices: *frame entry* and *frame exit*, which represent entry to and exit from the frame, respectively; *frame call constructors* and *frame call destructors*, which represent calls to constructors and calls to destructors of the class, respectively; *frame loop*, which facilitates sequencing of methods; and *frame call*, which represents calls to the public methods. Frame edges connect the vertices of the frame. A frame first calls class constructors, and then enters a loop that contains calls to each public method in the class. On each iteration through the loop, control can pass to any of the public methods. After the loop terminates, the frame calls class destructors.[8]

Framed Class Control Flow Graph

To perform analyses that require a control flow representation for the class, we expand the control flow graph for the frame, and add this expanded graph to the class control flow graph. The result is a framed class control flow graph. The steps required to create a framed class control flow graph are as follows:

1. Expand the control flow graph for the frame, as follows:

    - Split each frame call constructors node into a *frame call constructors* and a *frame return constructors* node, and adjust edges accordingly. For each constructor M in the class, add a *call M* node. Connect the frame call constructors node to each of the call M nodes with call edges.

[8]We initially presented the frame in Reference [5]. That version of the frame contained five vertices; constructors and destructors were called within the frame loop. However, because a constructor can only be called when a class is instantiated, we now place the call to the constructors outside the loop, and connected it to the frame entry; similarly, we now place the call to the destructors outside the loop and connected it to the frame exit.

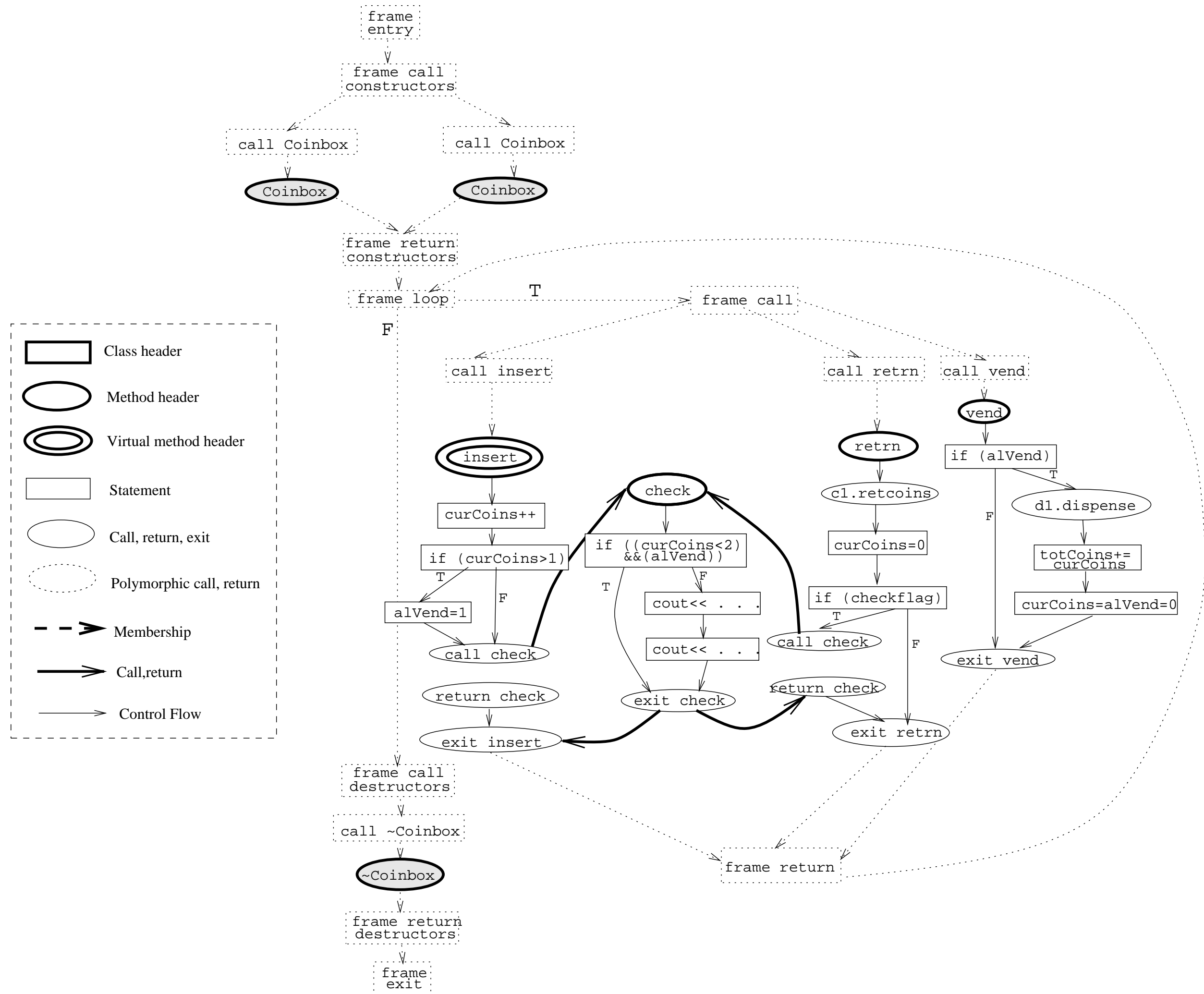

Figure 11: Framed class control flow graph for `Coinbox`; shaded nodes represent unexpanded portions of the graph.

- Split each frame call destructors node into a *frame call destructors* and a *frame return destructors* node, and adjust edges accordingly. For each destructor M in the class, add a *call* M node. Connect the frame call destructors node to each of the call M nodes with call edges.
- Split the frame call node into a *frame call* node and a *frame return* node. For each public method M in the class, add a *call* M node to represent a call to M, and connect it to the frame call node with a call edge.

2. Connect the expanded control flow graph for the frame to the class control flow graph by connecting each *call* M node to the method entry node in the class control flow graph for M, and connecting M's exit node to the appropriate frame return node.

Figure 11 shows a partial framed class control flow graph for the `Coinbox` class. The frame call constructors node has been expanded to represent calls to both `Coinbox` constructors, and the frame call destructors node has been expanded to represent a call to the `Coinbox` destructor. The frame call node is expanded to represent calls to each of the three public methods in `Coinbox`. All frame call nodes are attached to the control flow graphs for the methods they call.

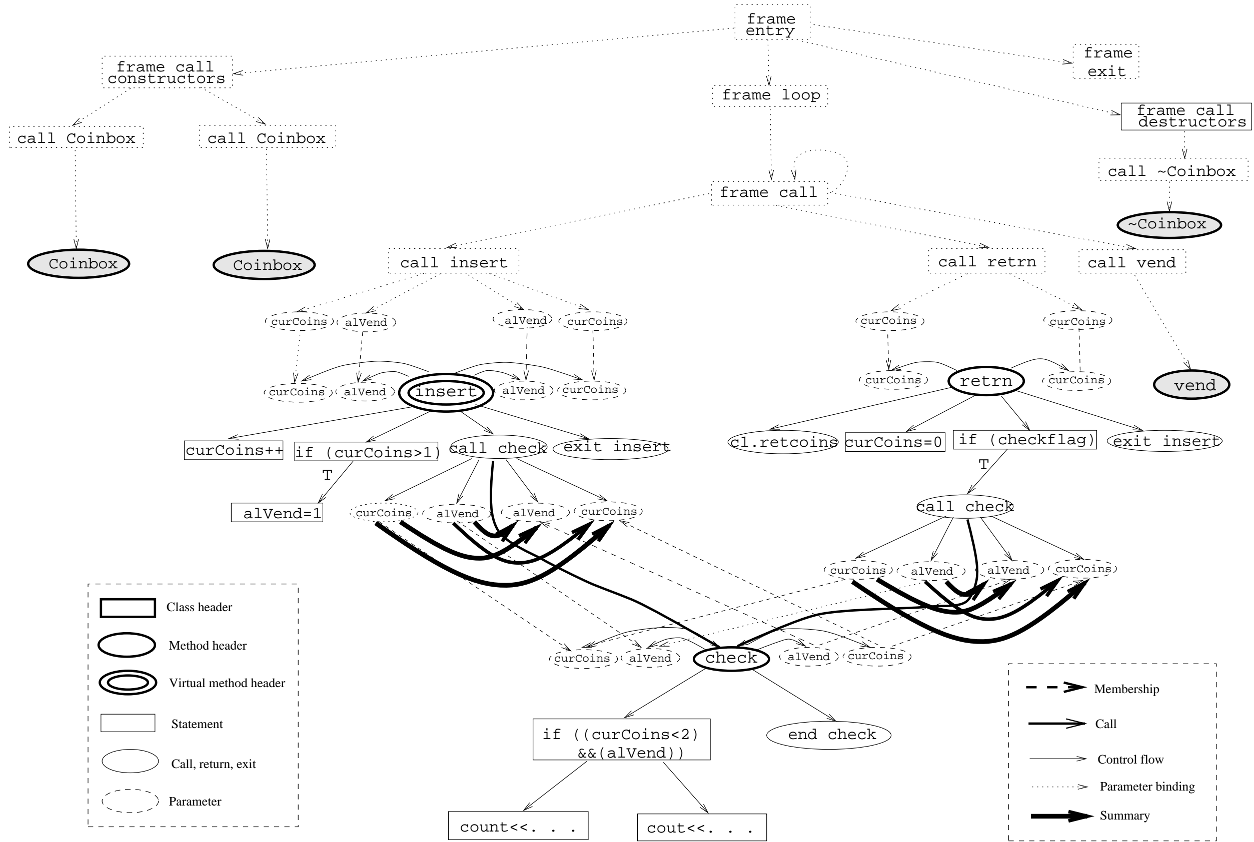

Figure 12: Partial framed class control dependence graph for `Coinbox`; shaded nodes represent unexpanded portions of the graph.

We construct a framed interclass control flow graph using a technique that is similar to the one described above, except that for these graphs we use interclass control flow graphs.

Framed class control flow graphs support direct use of flow-sensitive data flow and alias analysis techniques[5, 12]. The graphs thus facilitate data flow testing of classes[5].

Framed Class Dependence Graph

To perform analyses that require dependence information about the class, we expand the program dependence graph for the frame, and add this expanded program dependence graph to the class dependence graph. The result is a framed class dependence graph. The steps required to create a framed class dependence graph are as follows:

1. Expand the program dependence graph for the frame, as follows:

   - For each constructor M in the class, add a *call* M node. Add actual-in and actual-out vertices to match the formal-in and formal-out vertices of M, and connect these vertices to the associated call node with control dependence edges. Connect the frame call constructors node to each of the call M nodes.
   - For each destructor M in the class, add a *call* M node. Add actual-in and actual-out vertices to match the formal-in and formal-out vertices of M, and connect these vertices to the associated call node with control dependence edges.

- For each public method M in the class, add a *call* M node representing a call to M, and connect it to the frame call node with a call edge. Add actual-in and actual-out vertices to match the formal-in and formal-out vertices of M, and connect these vertices to the associated call node with control dependence edges.

2. Connect the expanded program dependence graph for the frame to the class dependence graph, by connecting each *call* M node to the method entry node in the class dependence graph for M; connect the actual-in and formal-in parameter nodes and the formal-out and actual-out parameter nodes with parameter edges. Finally, attach summary edges for the called methods at the call sites.

Figure 12 shows a partial framed class dependence graph for the `Coinbox` class. The frame call constructors node has been expanded to represent calls to both `Coinbox` constructors, and the frame call destructors node has been expanded to represent a call to the `Coinbox` destructor. The frame call node is expanded to represent calls to each of the three public methods in `Coinbox`. All frame call nodes are attached to the program dependence graphs for the methods they call. Parameter nodes that match the parameter nodes at the method entry to a called method are added to the call sites from the frame. Although the state variables represented by these parameter nodes may not be visible in the calling programs, the values of these variables must persist across calls to the method; these parameter nodes represent these persistent values.

We construct a framed interclass control dependence graph using a technique that is similar to the one described above except that for these graphs we use interclass control dependence graphs.

Framed class dependence graphs support direct use of interprocedural forward and backward slicing techniques[13], originally defined by Horwitz, Reps, and Binkley[7].

3 A system architecture

For the family of graph representations described in the previous section to be useful in practice, we must be able to construct and manage the representations efficiently. We should also be able to easily extend the family of representations to accommodate new analysis techniques and representations. In this section, we outline the architecture of an efficient, extensible system for constructing and managing our family of representations.

Data elements

Our system maintains two classes of data primitives: nodes and edges. Nodes and edges may be of various types; depending on its type, a node or edge may have specific information associated with it. At a higher level of abstraction, our system maintains *component sets*: these sets contain nodes, edges, and/or other component sets. A graph is simply a component set. This data organization lets our system reuse representational components, while efficiently distinguishing collections of related components. For example, most of the information relevant to a statement node may be stored with the node; component sets that contain the node need only identify the node, and do not need to rerecord the node's associated information.[9] Also, given this organization, if specific components are typically grouped together (e.g., the method header nodes for a specific class), we can associate these nodes with a component set S; then, any component set that requires these specific components as members need only include S as a member.

[9]For example, a component set may simply contain integer identifiers for (or pointers to) other nodes, edges, or component sets.

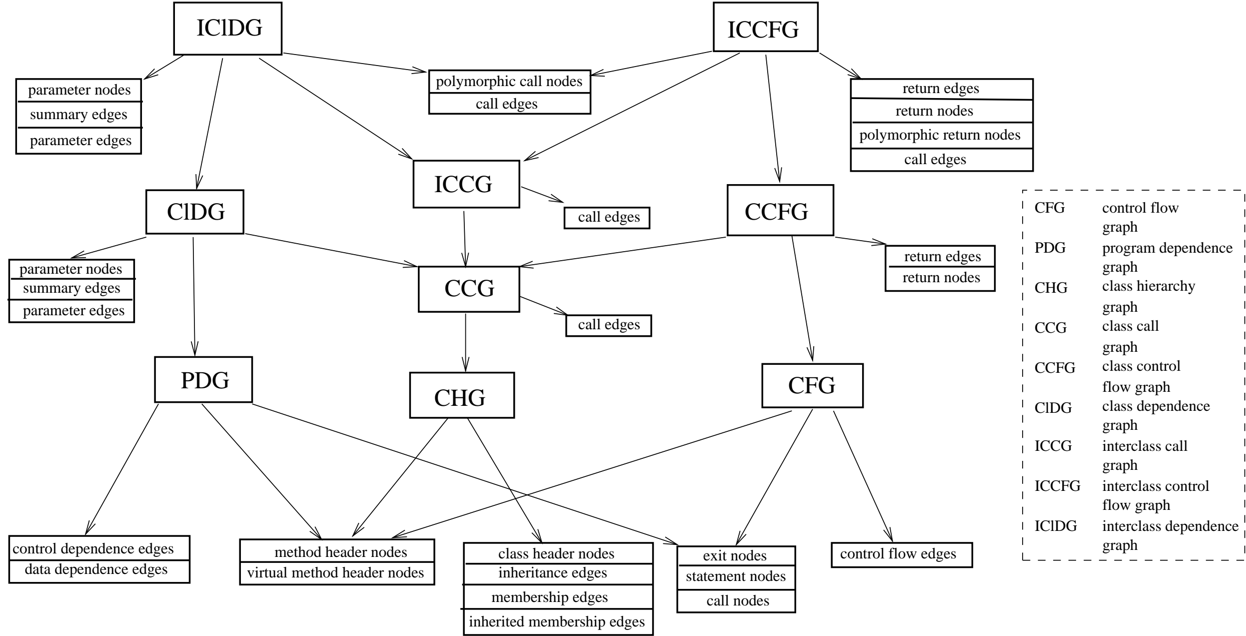

Figure 13: The organization of the data elements in our system.

Figure 13 depicts the organization of data elements required for the family of representations presented in Section 2. The figure omits individual nodes and edges, focusing instead on component sets and their relationships. In the figure, each rectangle represents a component set; edges indicate the component sets that compose various graphs.[10] For example, the figure shows that a class hierarchy graph (CHG) is composed of six component sets: method header nodes, virtual method header nodes, class header nodes, inheritance edges, membership edges, and inherited membership edges. A class call graph (CCG) is composed of a class hierarchy graph (CHG), and a set of call edges. An interclass call graph (ICCG) is composed of class call graphs and an additional set of (interclass) call edges.[11] The figure depicts an incremental data organization, and shows the extent to which that data organization supports reuse of representational components. For example, the figure shows that a single set of statement nodes serves as an element of the control flow graph (CFG) G for some method M, for the program dependence graph (PDG) for M, and for any class control flow graph (CCFG), interclass control flow graph (ICCFG), class dependence graph (ClDG), or interclass dependence graph (IClDG) that contains G.

This data organization supports system extensibility. For example, if we wish to add the class control dependence graphs required by the regression test selection algorithm of Reference [15] to our family of representations, we can easily build these graphs from component sets already defined for our family of representations. Alternatively, if a new analysis tool requires component sets not yet recognized by our system, we can add these sets, and combine them with existing component sets to derive new graph representations.

We have not depicted our framed graphs in Figure 13; these graphs simply group component sets for frames with component sets for class or interclass representations.

[10] More precisely, the figure displays an *is-composed-of* relation[3] over the universe of graphs and component sets that are required for our representations.

[11] Our figure does not distinguish cases where a component set is composed of exactly one instance of another component set from cases where a component set is composed of one or more instances of another component set.

Processing elements

Figures 14 and 15 depict the processing elements of our system, and relationships between those elements. Rectangles in the figures represent graph constructors; edges represent relationships between constructors.

Figure 14 is a data flow diagram of our system; as the diagram shows, we employ a data-centered architecture. With the exception of three constructors that take source code as input, and framed graph constructors that output framed graphs to analysis tools, all constructors retrieve data from, and deposit data into, a shared data repository. This architecture is advantageous because it supports reuse of representational components.

Figure 15 depicts ordering constraints among the graph constructors; a dotted line from constructor A to constructor B indicates that constructor A depends on (uses, or requires prior invocation of), constructor B. For example, the program dependence graph (PDG) constructor depends on the control flow graph (CFG) constructor, and the class control flow graph (CCFG) constructor depends on the control flow graph (CFG) and class call graph (CCG) constructors.

To obtain graph G, a graph constructor F first ascertains whether G is already present in the data repository, and if so, simply returns G. If G is not present in the repository, F invokes the graph constructors on which it depends. These constructors ensure that the graphs for which they are responsible are present in the data repository (by building them, or discovering that they are already present), and then return control to F. F then uses components now present in the data repository to construct G, deposits any nodes, edges, and component sets required for G including the component set for G itself, to the data repository.

Consider, for example, the control flow graph (CFG) constructor. This constructor is a primitive constructor in the sense that it is not dependent on other constructors. Called to obtain a control flow graph for method M, the control flow graph constructor first ascertains whether the graph is already present in the data repository, and if so, returns it. If the graph is not available, the constructor builds it, and deposits several types of objects in the data repository. At a primitive level, the constructor deposits control flow edges, and various types of nodes (statement, exit, and call) into the data repository. The constructor also deposits component sets of these nodes and edges into the repository. The constructor must also identify a method header node for M; such a node could already exist in the data repository if it had been created by a previous invocation of a class hierarchy graph constructor. The constructor initially seeks a method header node in the data repository and if it finds one, reuses it. If the constructor does not find such a node, it creates one. Finally, the constructor outputs a component set that represents the control flow graph itself.

As a second example, a class control flow graph (CCFG) constructor takes the name of a class C as input. The constructor first determines whether the class control flow graph for C is already present in the data repository, and if so, returns it. Otherwise, the constructor invokes the class call graph (CCG) constructor with C; that constructor ensures that the requisite class call graph is present in the data repository. Next, the constructor uses the class call graph to identify the individual control flow graphs that it requires. For each such control flow graph, the class call graph constructor invokes the control flow graph (CFG) constructor; that constructor ensures that the required control flow graph is present in the data repository. When all prerequisite representational components are available, the class control flow graph constructor assembles its graph, and deposits graph data into the repository. This data includes return edges and return nodes, component sets that contain these nodes, and a component set that contains the various components of the

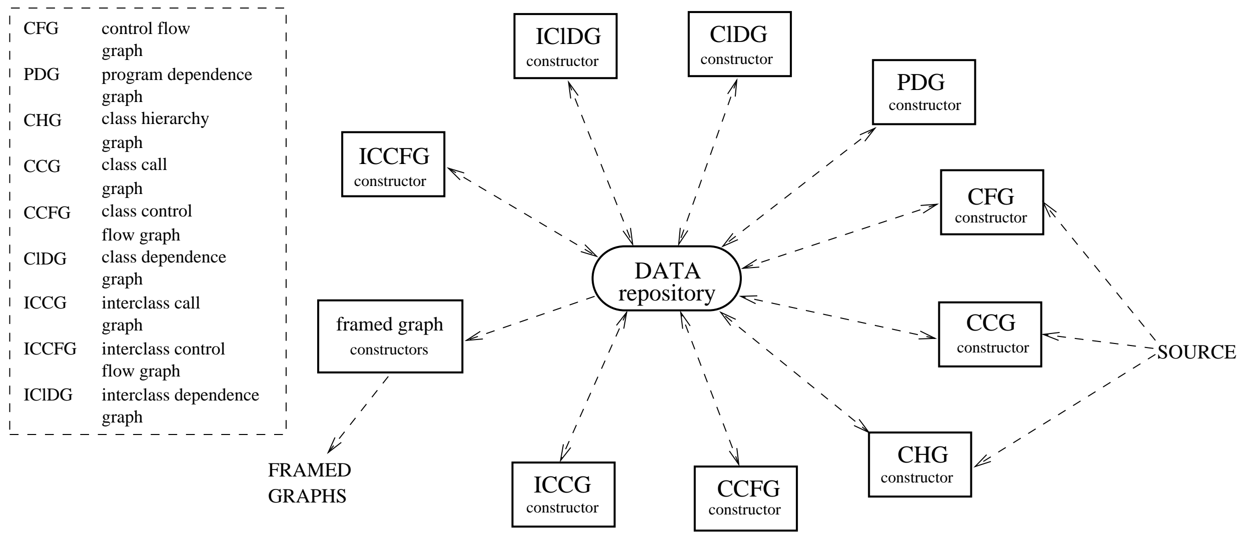

Figure 14: Data connections between processing elements.

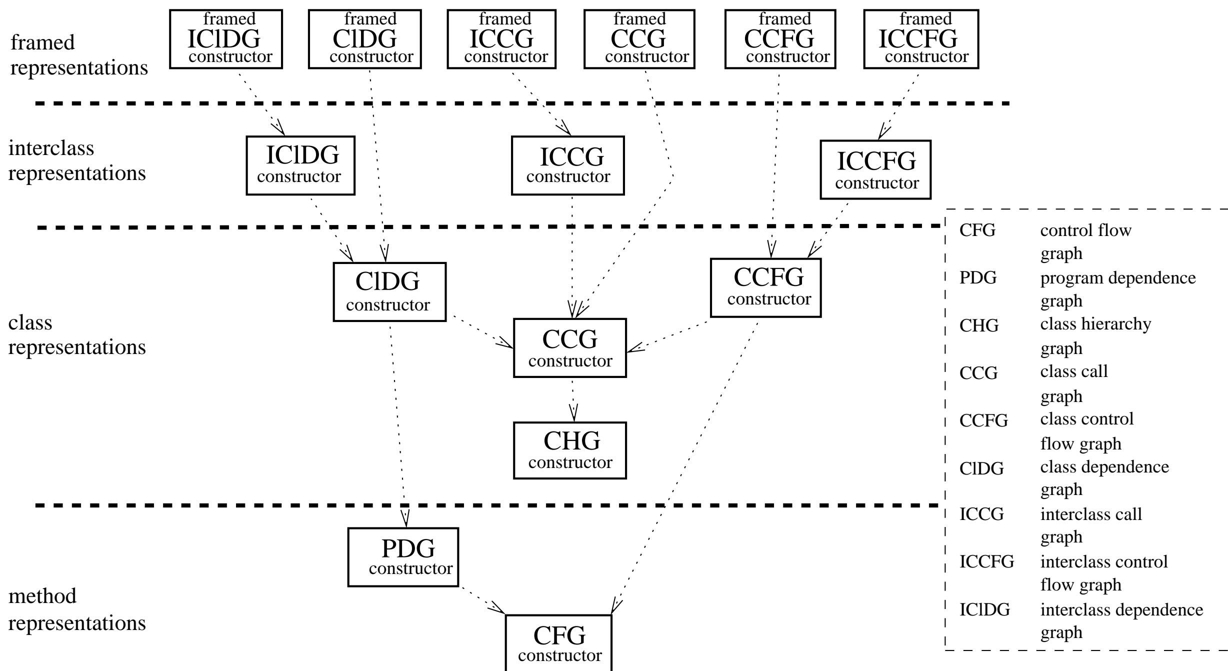

Figure 15: Ordering constraints on processing elements.

class control flow graph.

Other graph constructors follow similar steps. We omit details for reasons of space.

One potential source of inefficiency for a system designed from our architecture involves output to and input from the data repository. A system may achieve better performance in graph construction by maintaining a layered data repository, in which data is maintained (cached) in main memory as it is constructed or retrieved from disk, rather than being stored as each subgraph required by a higher-level graph is constructed. This functionality can be encapsulated within the functionality of a database management system, and thus hidden from graph constructors.

Advantages of our architecture

Our architecture has several advantages. The architecture supports a demand-driven approach to graph construction that lets a user obtain a representation at any level by making a single request. However, behind the scenes, the architecture supports the reuse of representational components, amortizing the cost of constructing and storing individual representations. The architecture provides for system extensibility; we can add new representations to our family by identifying the data elements that they can reuse, and the processing components on which they depend.

4 Related Work

Kung et al.[8, 10, 9] present a representation for use in testing object-oriented software. Their model consists of two components that are related to our work: an object relation diagram and a block branch diagram. The object relation diagram models the relationships, such as inheritance, aggregation, and association, that exist between classes, and provides static structural information about the relationships between object classes. The block branch diagram contains the control flow graphs of each class method, and the class's interface, and gives a static implementation view of the object-oriented program. Like the object relation diagram, our class hierarchy graph models inheritance relationships among classes, and our call graphs provide a static view of the relationships between object classes. However, our implementation view of the software differs from the block branch diagram in several ways. First, in addition to providing control flow information about individual methods, our control flow graphs provide interprocedural control flow information that represents the control flow of interacting methods. Thus, our representations facilitate analyses, such as flow-sensitive interprocedural data flow analysis and data flow testing, not supported by the block branch diagram. Second, our family of representations also provides control dependence information about the software that is not provided by the block branch diagram. Thus, our graphs facilitate analyses such as slicing. Finally, our family of representations provides a mechanism that facilitates separate analysis of classes, a feature not provided by the block branch diagram.

Malloy et al.[14] present a representation for object-oriented programs called an object oriented program dependence graph. Like our representations, the object-oriented program dependence graph represents control flow, data dependence, and control dependence. However, our representations differ from theirs in several ways. First, the object-oriented program dependence graph is defined only for complete programs: it does not provide a mechanism analogous to our frame that facilitates separate analysis of classes. Second, the authors attempt to reduce the complexity of their representation by omitting parameter nodes and binding

edges, and using fewer edges to represent polymorphic calls. The authors claim that this approach reduces the complexity of their graphs, and makes their representations "uncluttered by unnecessary nodes and edges". This improvement in graph appearance, however, is achieved at considerable cost. The resulting graphs, although visually appealing, require no less storage than ours (and in many cases require more storage), because they shift the burden of storage from that of providing nodes and edges to that of annotating other edges with textual information that can require more space to encode. Furthermore, the nodes that the authors eliminate from their graphs are *not* "unnecessary": in the absence of these nodes, the resulting graphs do not directly support the use of existing algorithms for data flow analysis and slicing. To perform data flow analysis and slicing on the object-oriented program dependence graph we need to modify the existing analysis algorithms; the modified algorithms are considerably more complex than the existing algorithms. In practice, we typically do not intend representations for object-oriented software to be viewed by humans, because for nontrivial programs these representations are too complex to be comprehended. Our goal in designing representations is that they be easily "viewable" by analysis tools. Our graphs achieve this goal. Moreover, if we require a view of a representation suitable for use by humans, viewing tools can easily and efficiently produce such a view from our graphs.

5 Conclusions

We have presented a coherent family of analyzable graph representations for object-oriented software. Our representations account for object-oriented language features such as inheritance, scoping, polymorphism and dynamic binding. Our family of graphs represents class hierarchies, calling structures, control flow, data flow, and control dependence, at both the class and interclass level. Also included in our family of graphs are representations for framed graphs, which facilitate the direct application of existing interprocedural analysis techniques, such as data flow analysis and slicing, to our representations. We have also described the architecture of an efficient, extensible system for constructing and managing our representations.

We are implementing a system for representing C++ software based on the architecture described in Section 3. Thus far, our implementation creates control flow graphs and program dependence graphs for individual methods, and class hierarchy graphs and class call graphs. We have used our implementation to gather metrics for C++ classes[17].

In this paper, we have focused on analyzable representations for classes; however, a system for representing object-oriented software should also represent applications programs that use classes. Our family of representations is easily extended to provide the necessary graphs. For example, to build an interprocedural control flow graph for an applications program, we build individual control flow graphs for the program's procedures, and wherever the program invokes methods, we reuse portions of class control flow graphs found in the database.

Acknowledgements

This work was partially supported by grants from Microsoft, Inc. and Data General Corp., and by NSF under Grant CCR-9357811 to Clemson University and Ohio State University. Manvinder Singh implemented portions of our analysis system[17].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison–Wesley Publishing Company, 1986.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[4] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. *Proceedings of the 14th International Conference on Software Enginering*, pages 68–80, May 1992.

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

[6] M. J. Harrold and M. L. Soffa. Selecting data for integration testing. *IEEE Software*, pages 58–65, March 1991.

[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. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. Design recovery for software testing of object-oriented programs. In *Working Conference on Reverse Engineering*, pages 202–211, May 1993.

[9] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object-oriented software maintenance. In *International Conference on Software Maintenance '94*, pages 202–211, September 1994.

[10] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994.

[11] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In *COMSAC 94*, 1994.

[12] 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.

[13] L. Larsen and M. J. Harrold. Slicing object-oriented software. *18th International Conference on Software Engineering*, pages 495–505, March 1996.

[14] 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.

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

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[17] M. Singh. A system for representing object-oriented software. Master's thesis, Clemson University, 1995.