A Coherent Family of Analyzable Graphical Representations for Object-Oriented Software

Mary Jean Harrold and Gregg Rothermel
Department of Computer and Information Science
Ohio State University
395 Dreese Lab, Columbus OH, 43210
{harrold,grother}@cis.osu.edu

Abstract

Many software engineering tools and techniques rely on graphical representations of software, such as control flow graphs, program dependence graphs, or system dependence graphs. To apply these tools and techniques to object-oriented software, we cannot necessarily use existing graphical representations of procedural software. Representations of object-oriented software, like those for procedural software, must depict individual procedures (methods) and entire programs; however, they must also depict classes and their interactions, and account for the effects of inheritance, polymorphism, and aggregation. These representations should facilitate the use of existing program analysis tools and techniques, rather than requiring us to create 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. In this paper, we describe a coherent family of graphical representations of object-oriented software that meet the foregoing qualifications, and we outline the architecture of an efficient, extensible system for constructing and managing those representations.

1 Introduction

Many software engineering tools and techniques require program analysis information. For example, data flow testing and program slicing require data flow analysis information, and techniques for debugging, regression testing, and impact analysis use program slices. These techniques typically function on graphical program representations; to apply them to object-oriented software, we must first construct appropriate representations for that software.

Representations for procedural software, such as control flow graphs, program dependence graphs, and system dependence graphs, 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. In particular, representations for object-oriented software must efficiently and effectively model classes, objects, inheritance, scoping, polymorphism, and dynamic binding.

Classes are the primary component of object-oriented systems. 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 independent of calling environments, with the intention of supporting reusability. Representations for classes should be independent, analyzable, and reusable.

Derived classes are created through a mechanism known as inheritance. Inheritance permits a derived class to inherit (reuse) attributes from its parent classes, and extend, restrict, redefine, or replace them in
some way. Just as inheritance facilitates code reuse, it can provide an opportunity for reuse of representational components. Class representations should exploit this opportunity.

Although the visibility (scope) of an object’s instance variables (state) is limited, these variables retain their values between calls to objects. A representation for object-oriented software must account for the persistence of instance variables across calls, in a manner that facilitates proper calculation of dependencies across those calls, even though those instance variables may not be visible outside of the object’s own methods.

Polymorphism permits a choice at runtime (by dynamic binding) of one of a possible set of destinations of a method call. To statically represent a dynamic choice, a representation must account for all possible bindings, unless we can statically determine a precise or reduced set of bindings.

Finally, in addition to accounting for the foregoing features, representations for object-oriented software should support the use of existing program analysis tools and techniques wherever possible, rather than requiring the creation of new techniques.

In this paper, we present a coherent family of graphical representations for object-oriented software that meets the requirements outlined above. Our representations model classes, inheritance hierarchies, calling relationships, control flow, control dependence, and data dependence. We also provide representations for classes and interacting classes that facilitate their analysis as independent entities, in the absence of specific calling environments. Thus, our representations support analysis of object-oriented software at the class, interclass, and applications levels, and handle the features of object-oriented languages. Moreover, our representations support the direct application, without modification, of existing algorithms for data flow analysis and slicing; thus, our system supports the use of existing techniques for regression test selection, data flow testing, impact analysis, and metrics collection that depend on those algorithms. We outline the architecture of a system for constructing and managing our representations that facilitates the reuse of representational components, and promotes system extensibility.

In the next section, we describe our family of graphical program representations. In Section 3 we outline the architecture of a system for building and managing these representations. Section 4 discusses related work on representations for object-oriented systems, and details the advantages of our approach. Finally, in Section 5, we present conclusions and discuss future work.

2 A Family of Graphical Representations

In this section, we describe our family of 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 \textit{class level} for single classes or classes that are related by inheritance. Finally, we show how to extend the representation to the \textit{interclass level} for interacting 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
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);
        }
    }
};

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::retrcoins(unsigned number) {
        cout << "User: Take your " << number << " coins" << endl;
    }
};

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: User can vend for free!" << endl;
    }
}

class Coinbox {

}
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 graph4 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 coinsinwindow method simulates the display of the number of coins by outputting a message. Figure 2 shows the class hierarchy

3 Instead of defining two constructors for Coinbox, one with checking code and one without checking code, we could have passed a parameter to set the flag and used only one constructor. We use two constructors to illustrate our representations for classes with more than one constructor.

4 Some of our node and edge naming conventions are similar to those used in Reference [14, 15].
class Changebox : public Coinbox {
    public:
        Coinbox::Changebox::Changebox() :
            Coinbox() {}
        Coinbox::Changebox(unsigned flag) :
            Coinbox(flag) {}
        Coinbox::<Changebox() {}
    void Changebox::insert()
        if (curCoins > 1)
            cl.returncoins(1);
        else
            Coinbox::insert();
    void Changebox::shocoins()
        d3.coinsinwindow(cur);
}    
class Display {
    public:
        Display::Display() {}
        Display::~Display() {}
    void Display::
        coinswindow(unsigned number) {
            cout << "User: You have inserted "
                number << " coins" << endl;
        }
    protected:
        Display d3;
};

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

A class hierarchy graph has various uses. For example, one class testing technique[3] 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, depth and width of inheritance hierarchies, and numbers of friend functions and friend classes.

2.2 Class Call Graphs

A call graph represents static calling relationships among procedures. A class call graph represents these 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 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). By inspecting the implementations of Coinbox methods, we discover that 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. (*) MHH: is the following correct? Does the figure show the named edges? *)
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 cl 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 certain 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.\(^5\) If the called method is not in the hierarchy, we postpone splitting the
call node into call and return nodes until we want to connect the node with the class call graph of the called
method, and create an interclass control flow graph of the called class.\(^6\)

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.

\(^5\) In this sense, our graphs resemble the interprocedural control flow graphs defined originally by Landi and Ryder[11].

\(^6\) In Reference \[4\], 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.
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 graph also includes nodes and edges that represent control flow for methods newly defined for the derived 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 Vending may have (*MJH: is “have” the right term?*) 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 control flow graphs directly support the use of flow-sensitive data flow and alias algorithms[11]. (*) MJH: Can you think of other uses? * The graphs thus facilitate data flow testing of interacting methods[5].

2.4 Class Dependence Graphs

Class dependence graphs are based on the system dependence graphs initially defined by Horwitz, Reps, and Binkley[6]. 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 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.

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.

7 Although called a “program dependence graph” for historical reasons, these graphs actually represent dependencies for single procedures.

8 Although there are other types of data dependence, we consider only flow dependence.
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 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 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[6], that can be applied to the interacting methods in the class. Also, the regression testing technique of Reference [16] uses a similar graphical 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 representation. 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 analysis 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.
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.9

Framed Class Control Flow Graph

To perform analyses that require a class control flow graph, 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.

   9We initially presented the frame in Reference [4]. 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.
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[4, 11]. The graphs thus facilitate data flow testing of classes[4]. (* MJH: I think you mentioned the fact that really the analysis techniques we describe work on the framed graphs, not on the ones presented earlier. Thus was it a mistake to say what those graphs were good for earlier? Or should the uses of the graphs be discussed in a section of its own, after this frame section? Might not be time to mess with this. *)

**Framed Class Dependence Graph**

To perform analyses that require a class dependence graph, 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; these nodes are only place holders for call sites in applications programs.

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.

Class dependence graphs support direct use of interprocedural forward and backward slicing techniques[12], originally defined by Horwitz, Reps, and Binkley[6].

3 A system architecture

For the family of graphical 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 individual nodes, edges, or 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.10 Also, given this organization, if specific components are typically grouped together (e.g., the method header nodes

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

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.\footnote{More precisely, the figure displays an *is-composed-of* relation over the universe of graphs and component sets that are required for our representations.} 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), in turn, 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.\footnote{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.} 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 (CIDG), or interclass dependence graph (ICIDG) 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 [16] 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 graphical...
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.

**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 dataflow 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
Figure 14: Data connections between processing elements.

Figure 15: Ordering constraints on processing elements.
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 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

Malloy, et al.[13, 15] present a unified 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 dataflow 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. The opposite task — that of achieving analyzable “views” of the object-oriented program dependence graph — is not as straightforward, and requires substantial effort. 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.

Kung et al.[7, 9, 8] present a representation for use in testing object-oriented software. Their model consists of three components: an object relation diagram, a block branch diagram and an object state 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 for testers and software maintainers. 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. The object state diagram represents the behavior of an object with respect to its (* MJH what here? *). Although these graphs are useful for some types of analysis, the information they provide is not sufficient to facilitate the types of analysis that our system supports. (* MJH: more? *)

5 Conclusions and Future Work

We have presented a family of graphical 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.

(* MJH: I stopped here. *)

Our family of graphs can be easily accessed by applications programs, although we did not discuss it in this paper.

Our architecture provides ... Our construction ... Our system is extensible, and other types of graphs can easily be added to our system. For example, the system can be easily extended to create ... Mention that there are other types of graphs we can easily get given our architecture. Are there other tools or techniques that we could support, given other graph types?

Our implementation of the first levels of our system[17] for C++ creates control flow graphs and program dependence graphs for individual methods in a class, and class hierarchy graph and a class call graph for the classes that interact thorough inheritance. Thus, a system for representing classes must provide a means to analyze a class independent of its calling environment.

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.

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 a version of our analysis system[17].
6 NOTES

- **Title.** Try it without the word “coherent”. Or without the phrase “a coherent family of”. Or without the word “analyzable”. Like any of those better?

- **General.** I suppose, properly speaking, we should be talking about sending messages rather than having method calls. Occasionally (section 2.2, last two paragraphs) we do use the “message sending” terminology. But the message sending terminology is very awkward. Can we get away with the “call” terminology? Do we need a footnote to say that we’re not dummies but are using the call term because, well, because why?

- **Miscellaneous.** Grep through the .tex files for the string “MJF’. I’ve embedded a couple of short notes in places where it was easier than recording things here.

- **Definitions.** Here are a couple I’ve refined, I hope not incorrectly. By definition, the intraclass level contains graphs that are intra-hierarchy only. The interclass level contains graphs that connect hierarchies. We may choose to expand only some connections, but if we show any interclass connections at all we’re at the interclass level. The intraclass level contains what we call ”class” graphs: CHG, CCFG, CIDG, CCG. The interclass level contains what we call ”interclass” graphs: ICCFG, ICCG, etc. When we depict a class graph in a figure, if we omit, for brevity, a graph portion that stands for something in the class, we are showing a partial graph. But leaving a call site that calls out of the hierarchy unexpanded does not give us a partial graph when dealing with class graphs, because those sites are always unexpanded in class graphs. So the statement ”the figure depicts the class graph ignoring interactions with classes outside the hierarchy” is redundant. By definition, a class graph necessarily avoids interactions with classes outside the hierarchy.

- **Section 1, paragraph 2.** The word ”objects” appears in the list at the end of the paragraph. I don’t see us doing anything particular about objects in the rest of this particular paper though. Should we remove the term from the list? The list also used to (in my first draft) mention aggregation but we don’t seem to be covering it so we should leave it out or say more about it, even if only in our conclusions. Note that the term aggregation appears in the abstract.

- **Section 1, paragraph 5.** The paragraph is true, and I believe it’s the discussion of what “scope” in the list 3 paragraphs above meant; but do we in fact handle this in our representations? We may support the handling of it, but do we discuss how we support it in this paper? (You say we do handle this, so we need to say later in this paper that we’re handling this problem.)

- **Section 1, paragraph 7.** Doesn’t this raise the spectre of complicated graphs for languages like Smalltalk? Do we need to mention this somewhere? Conclusions?

- **Figure 2.** Most figs have the retrn node under Coinbox at the right, and check to left of center. One of the graphs in this figure has them reversed. Swap node labels.

- **footnote 2.** Is this really necessary? If it is necessary (e.g. to keep the reader from thinking we are single-lobed programmers), we might abbreviate the footnote to something like: “We could also
implement this checking functionality using a single constructor; we use the two simply for purposes of illustration.”

- **footnote 3.** Ugh. Must we? Did we borrow these names, or did we come up with them ourselves because they are incredibly obvious (I forget which ones of ours they also use). Also, did you originally use the names before they did, such as in the HIT paper? If so, they borrowed ours, and citing the similarity is not necessary. I just hate to draw more attention to them than absolutely necessary.

- In some places, even though a figure had a legend, the text in the paper also said things like “bold rectangles represent the starship Enterprise, and dotted lines represent the paths followed by Klingon warships after coming out of warp drive.” Since it’s in the legend maybe it isn’t necessary? I’ve just commented some of these out in the .tex files so it’ll be easy to restore them.

- Figure 11 is missing its legend.

- Should c1.xxx and d1.xxx nodes be shaded in class graphs? They aren’t exactly “unexpanded portions of the graph” – they aren’t supposed to expanded in class graphs.

References


