A Comprehensive Program Representation of Object-oriented Software [1]

John D. McGregor
Brian A. Malloy
Rebecca L. Siegmund
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906
johnmc@cs.clemson.edu

Abstract

An extensible representation for object-oriented programs, the Object-oriented Program Dependence Graph (OPDG), is presented. The representation is divided into three layers: a first layer that presents the class inheritance structure, a second layer that combines a traditional control dependence subgraph and a data dependence subgraph with objects, and a third layer that presents the dynamic, runtime aspects of an object-oriented program as an Object Dependence Subgraph. The representation is modular allowing specific tools to only use the portion required for the tool's operation. The complete representation provides information sufficient for most program analysis techniques including data flow analysis, reverse engineering, interactive debuggers and other tools.

1 Introduction

Object-oriented software presents unique opportunities and problems for program representation and analysis schemes. The goal of the research reported in this paper is a program representation for object-oriented programs. An outline of the representation, the Program Dependency Graph for Object-oriented Systems (OPDG), was originally presented in [18]. In this paper we present the complete representation and illustrate its capabilities with example applications.

The representation is composed of three layers, the Class Hierarchy Subgraph (CHS) representing the class inheritance structures; the Control Dependence Subgraph (CDS) and the Data Dependence Subgraph (DDS) providing the static, compile-time semantics of the program; and the Object Dependence Subgraph representing the runtime configuration of objects in the program. The OPDG explicitly represents object-oriented concepts such as inheritance, inclusion polymorphism and dynamically bound messages.

An incremental approach, that adheres to object-oriented design principles, is used to build the representation. The representation can be built for a single class or a complete program. The representation for a class is begun by reusing the representation of its parent classes to allow a more compact representation of the program as a whole. The layered architecture of the representation also supports the use of only those portions that are required for the analysis being conducted in a specific application.

The OPDG incorporates or adapts several existing techniques for representation of procedural programs including the Program Dependence Graphs. For example, the concept of data flow is adapted to represent the flow of objects, both statically and dynamically, in object-oriented programs.

[1] This research was partially supported by a grant from COMSOFT, IBM and BNR.

The comprehensive representation, the OPDG, described in this paper contains sufficient information to support a range of activities that use program analysis techniques. Control and data dependence information are used to select test data, determine test set adequacy [16], extend data flow testing techniques [9], generate reduced test sets for programs, implement regression testing and support debugging [11] [4] [14]. Since data dependence information explicitly represents the definition-use relationships implicitly present in a source program, it is useful for metrics, vectorization techniques and compiler optimizations.

This paper presents several contributions:

1. A three layered comprehensive program representation of object-oriented software;
2. New meanings of *definition(def)* and *use* for analyzing object-oriented software;
3. New concepts, termed *object flow* and *object dependence*, that support the analysis of object-oriented programs; and
4. Applications of the representation that include program slicing and metrics.

The comprehensive representation described in this paper has been designed with the following attributes. The representation should be:

- Easy to understand,
- Language independent,
- Able to support a range of program analysis activities and tools,
- Extensible to new language features, and
- Compact when compared with existing program representations.

In the next section we provide a review of related and supporting work including a discussion of Program Dependence Graphs and other program representation schemes. In section 3 we present a comprehensive definition of the representation. In section 4 we discuss the prototype of the representation and in section 5 we consider an application of the OPDG. In section 6 we discuss future work.

2 Background

2.1 Object-Oriented Systems

We assume that the reader is familiar with the basic concepts of object-oriented software. A discussion of these concepts may be found in the following references [5, 17, 24, 30].

2.2 Program Dependence Graphs

A PDG encodes control dependencies in a *control dependence subgraph* (CDS) and data dependencies in a *data dependence subgraph* (DDS). The CDS and DDS together provide an integral view of the important dependencies that facilitate those types of operations that require a view of the program that transcends basic block boundaries[2].

The PDG has proven useful for program analysis at all stages of the compilation process. The PDG can be constructed from intermediate code or from assembly code to facilitate aggressive program optimizations[27], register allocation [25], vectorization[3], and instruction scheduling[10, 21]. In addition, the PDG can be constructed from the source code or from the abstract syntax tree[22] to facilitate software engineering applications such as program slicing[28], determining test set adequacy[15], generating reduced test sets for programs[11], and regression testing[1, 2, 4, 14, 29]. The program representation that we consider in this paper, the OPDG, is primarily a tool for software engineering and is constructed during the parse stage of compiling the program or some portion of the program.

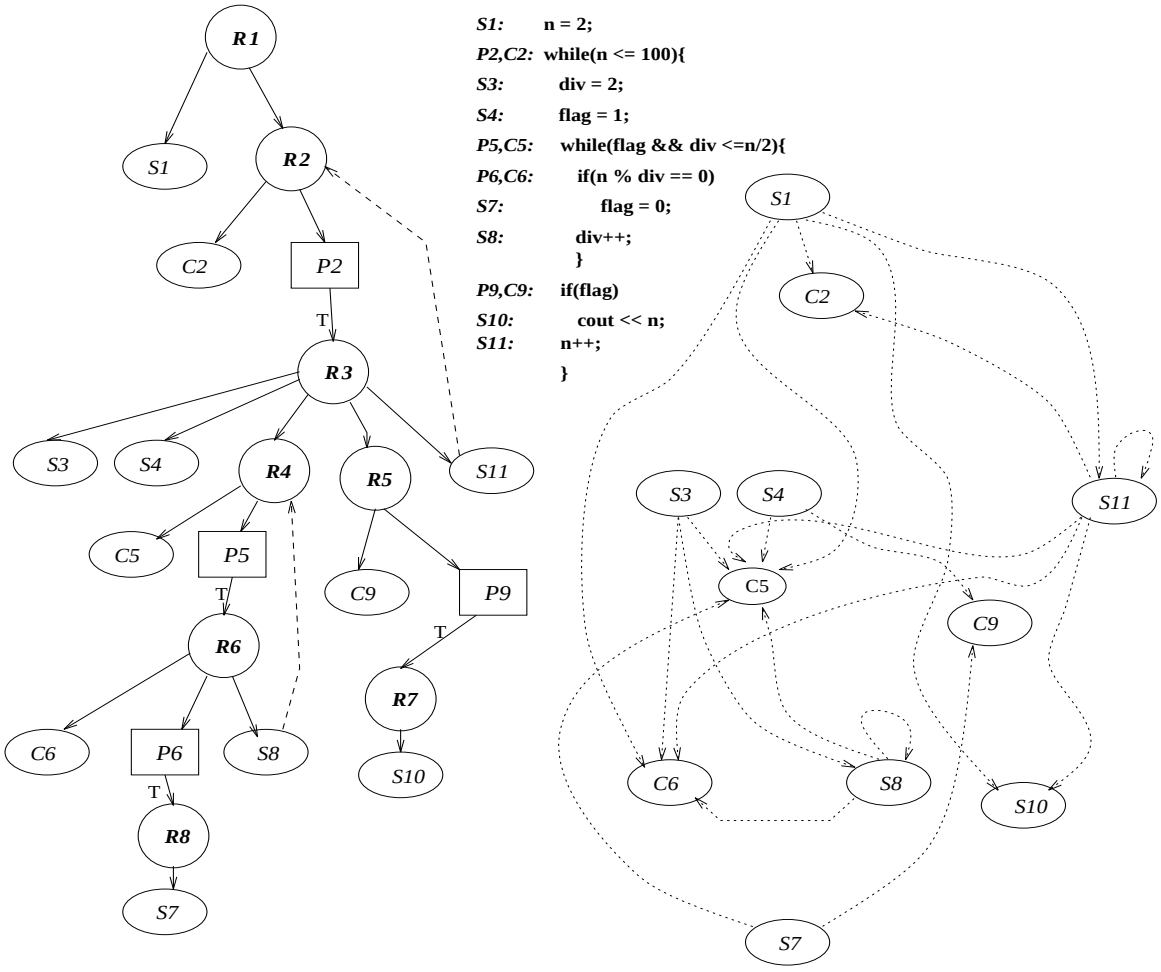


Figure 1: A program segment, its CDS (left) and DDS (right)

[2]A basic block is a sequence of code where the only entrance is through the first statement in the sequence and the only exit is through the last statement in the sequence.

The nodes in a CDS represent either single statements, or regions of code that have common control dependencies. A program segment and its corresponding CDS is illustrated in Figure 1. In the CDS of Figure 1, there are eight region nodes and eleven statement nodes where a statement node in the CDS corresponds to a single statement in the program segment. For our PDG, we follow the notation used in reference [22], so that circles represent region nodes, ellipses represent statement nodes and squares represent special region nodes called predicate nodes. Solid lines in Figure 1 represent control dependencies and dashed lines are used to represent control flow, e.g., loop backedges.

In Figure 1, region R1 represents the set of control conditions required for execution of the first statement in the program; region R1 consists of statement S1 and another region R2. Region R2 consists of statement C2 and region P2 where C2 is a *condition* statement and P2 is a predicate node representing the set of control conditions for region R3. Moreover, Region R3 is control dependent on P2-true.

The DDS for the program in Figure 1 is shown at the far right of the figure. The DDS is obtained by creating edges between nodes in the CDS to represent data dependencies; the DDS of Figure 1 shows flow dependencies resulting as data flows from a definition to a use. Anti- and output-dependencies can also be incorporated into the DDS. To illustrate the use of edges to represent data flow, consider that statement node S4 contains a definition of *flag* and nodes C5 and C9 contain uses of *flag* so that there is an edge from S4 to C5 and an edge from S4 to C9. Region nodes and predicate nodes are not included in the DDS since they do not contain definitions or uses of variables.

2.3 Other Program Representations

The PDG was originally developed for intraprocedural program analysis[27] and later extended for interprocedural program analysis[28]. Interprocedural program representations are variations of a program's *call graph*, where nodes in the graph represent the individual procedures and edges represent call sites. Each edge in the graph is labeled with the actual parameters associated with the call. Efficient construction of the call graph is described in reference [31]. Since the call graph represents the procedural calling sequences, it is a useful program maintenance tool and can also be used for interprocedural data flow analysis. However, interprocedural data flow analysis performed on the call graph is *flow insensitive* because the control flow of individual procedures is not considered[12].

To provide *flow sensitive* information, a variation of the call graph, the *program summary graph* (PSG)[6], is required. The program summary graph is sensitive to the flow of control across procedures and can provide information about reference parameters to permit parallelization of loops.

However, the program summary graph does not provide information about where, in a procedure, a reference parameter is defined or used. Information about definitions and uses of variables is required to test interfaces among procedures. The *interprocedural flow graph* (IFG)[13] extends the program summary graph to provide information about the locations of definitions and uses of reference parameters and global variables that can be reached across procedure boundaries.

While the IFG provides information to compute interprocedural data dependencies, it does not contain information about control dependencies in the program. This information is required for debugging and for detecting statements that can execute in parallel. The *system dependence graph* (SDG)[28] models both the control and data dependencies in the program and permits computation of interprocedural data flow information. The SDG combines dependence graphs for individual procedures[27] with additional nodes and edges to permit computation of an interprocedural slice

of the program.

The *unified interprocedural graph (UIG)*[12], combines the features of several existing program representations. The UIG provides control flow, data flow, control dependence, and data dependence information required for program maintenance. The information is extracted from the Call Graph, the PSG, the IFG and the SDG and eliminates redundancies in the individual graphs.

3 Program Representation

In this section we present the three layer representation. We first provide an overview of the representation. Next each layer is described in detail. We discuss how the imperative definitions of control flow, control dependence and data dependence in the PDG provide the foundation for the second and third layers of the OPDG. We then adapt these concepts to the interaction of objects and discuss the implications. We also discuss specific features of the representation for the handling of parameters and object attributes.

3.1 Overview

The OPDG is a comprehensive representation that depicts object-oriented software in a clear and concise manner. The representation is useful for a variety of applications since it provides a complete picture of the software system. The representation is easy to use because it contains standard program analysis structures such as control and data flow graphs and it is easy to understand because these structures are closely related to the object-oriented concepts embodied in the program.

The representation is in three layers. The first two layers provide a complete compile-time view of an object-oriented system while the third layer provides the runtime view. The union of the three layers is a Program Dependence Graph for Object-Oriented systems (OPDG).

The first layer is the Class Hierarchy Subgraph (CHS). The CHS is a graphical representation of classes, specifying the inheritance relationships between classes and the composition of methods into a class definition. The CHS for the system includes a *Class* header vertex for each class and a *Method* header vertex for each method defined in the class. Edges in the CHS also connect each *Class* header to corresponding headers of the other classes from which it inherits. Method headers are connected to the *Class* header for the class in which they are defined. Subclass representations do not have representations for methods defined in the superclass. The CHS does not represent any implementation details of the methods.

The second layer provides the implementation details of the methods of each class. This layer contains the Control Dependence Subgraph (CDS) and the Data Dependence Subgraph (DDS). This is a static representation so some information cannot be completely resolved, i.e., resolution of dynamically bound messages. The second layer represents polymorphism by identifying the set of methods that are possible polymorphic substitutes for the recipient of a message. Objects are added to the representation at this layer. Objects are handled atomically without regard to their internal structure. The traditional idea of data dependence is modified to provide a foundation for constructing the DDS containing objects.

The third layer is the Object Dependence Subgraph (ODS). This level of the representation depicts the interactions among objects. These interactions are the statically and dynamically bound messages between objects. This layer represents the dependencies that result from these messages. The dependencies are scoped so that the objects represented at any point in time are only those that are *live*. The layer also supports traversing the composition hierarchy of an object as well as the association relationships.

Because of the incremental nature of our representation, not all of the layers have to be built for the OPDG to be useful. Different tools may only need one or two of the layers. The first layer of our comprehensive representation provides support for metrics such as computing the number of classes in an object-oriented system or the average number of methods defined per class. The addition of the second layer to the representation provides support for a variety of static analyses such as detailed metrics, program analyses, and automatic test case generation. The addition of the third layer to the representation enables the computation of dynamic analyses that the first two layer cannot provide. Debugging tools and reverse engineering tools can be built from the information provided by the complete OPDG.

3.2 Syntax of the Representation

The basic diagrammatic notations for the different edges and vertices in the OPDG are illustrated in figure 2.

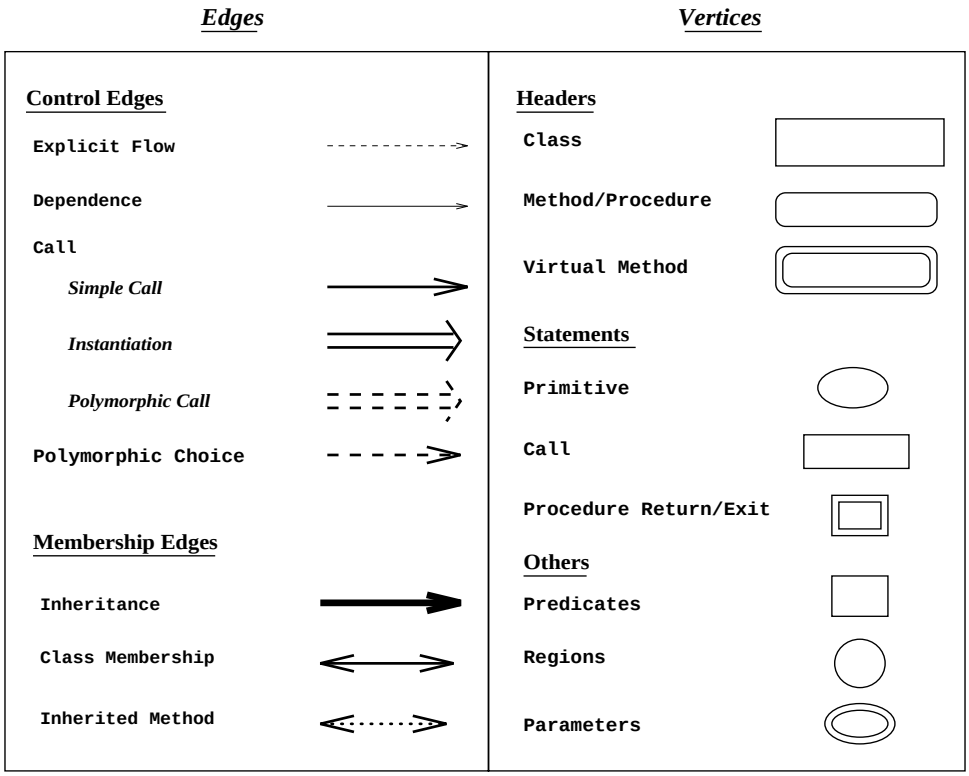


Figure 2: Syntactic notations for different edges and vertices

3.2.1 Vertices in the OPDG

The different vertices (see figure 2) used in our representation of an object-oriented program are:

- **Headers**
  Headers signify the beginning of a collection of representable entities with a syntactic analogy to the actual code.

– **Class**

A *Class* header signifies the grouping of all the methods and data attributes in the class. It also takes part in representing the inheritance hierarchy between the classes. The *Class* header contains a list of methods defined in the class, pointers to all the inherited methods, and links to any superclass(es) or subclass(es). Information regarding the data members of the class are also available in the *Class* header.

```
#include <iostream.h>

class String{
  public:
    String(const char *);
    virtual ~String();
    virtual void do_some_oprn_on_str();
    friend ostream&
         operator <<(ostream& os, String& s);
  private:
    char *str;
    };

ostream& operator<<(ostream& os, String& s)
{
 if (s.str==NULL)
    return os << "<empty>";
 return os << s.str;
}
```

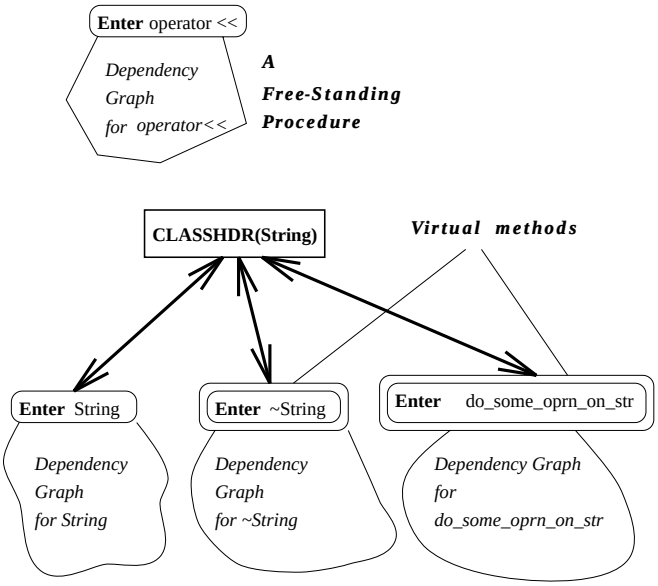


Figure 3: String class example in C++ with a free standing friend *operator*<< procedure

---

– **Method/Procedure**

A *Method* or a *Procedure* header is the entry vertex for the corresponding method or procedure. Its representation is similar to the entry vertex of a procedure as given in [22] or [28]. The *Method* header adds additional meaning to the representation by encapsulating information such as the class to which it belongs, data attributes of the class and parameter translation from actual to formal and vice-versa. The *Method* header also forms one of the end points of the *Class Membership* edge.

Free standing procedures are seen in a few object-oriented languages, and are not bound to any class (see figure 3). A header for such a *Procedure* is identical to the *Method* header but has no attachment to any *Class* header. The scope of the procedure is the program environment or the file at the least. The scope of the method is defined by its access privileges that indicate whether it is public or private.

– **Virtual Method**

Some object-oriented languages allow all the methods in a class to be dynamically bound. In contrast, C++ restricts the choice of methods that can be dynamically bound by having the keyword *virtual* before the method declaration. We use a similar approach to distinguish between the methods that can and cannot be dynamically bound. This helps understand the representation for polymorphism and dynamic binding easier.

A virtual method is a special member of a class, since a reference to this method, if polymorphic, would have to be resolved dynamically. As such, an enhanced representation is provided for the header of a virtual method. The *Virtual Method* header, along with

the *Polymorphic Choice* edge, plays a significant part in explicitly representing dynamic binding. Except for the additional features added to the representation for a *Virtual Method* header, all other features like class membership and class scoping resolution, are similar to that of a *Method* header.

- **Friend Function**
  Figure 3 gives the representation for an example String class in C++ and an associated friend *operator<<* procedure. In the representation, the *operator<<* procedure is a free standing procedure and is not attached to any class header. There is no special edge for a friend function.

- **Statements**

  - **Primitive**
    A *Primitive Statement* vertex represents a statement that carries out some part of a computation and does not result in either a call to a method or a procedure, or a return or exit from the procedure.
  - **Call**
    A *Call* vertex represents a call to the method or procedure at the call site. This is the starting point for the different call edges in the OPDG, namely, the *Simple Call* edge, the *Instantiation* edge and the *Polymorphic Call* edge. The different types of *Call* edges listed above are discussed in section 3.2.2.
  - **Procedure Return or Exit**
    A *Procedure Return* or *Exit* vertex signifies the last statement to be executed in that control flow path.

- **Others**

  - **Predicates**
    A *Predicate* vertex represents the predicate part in a conditional statement.
  - **Regions**
    A *Region* vertex groups statements with the same control dependencies.
  - **Parameters**
    Parameter vertices are used to signify both actual and formal parameters. For every in_parameter, actual and formal, we have one *Parameter* vertex. If the parameter is such that it may-be-modified, then we have a *Parameter* vertex each for the formal and actual out_parameters. Parameter vertices are discussed further in section 3.3.2.

3.2.2 Edges in the OPDG

A list of the various edges in our representation is given in figure 2. A brief description of each type of edge is given below.

- **Control Edges**

  - **Flow Edges**
    These are uni-directional edges that depict the explicit flow of control in the program. Implicit flow is denoted using the left-to-right notation of [22]. In figure 4, for the example code segment given, implicit flow of control is represented between the statements *S20, S21* and *S22* using the left-to-right notation. A few instances of explicit control flow are

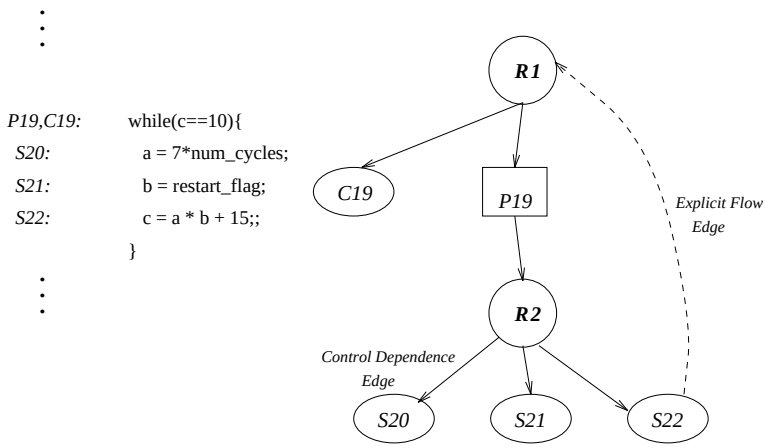


Figure 4: Example representation for Control Flow and Control Dependence Edges

return to the beginning of a while-loop, an *exit* or a *break* statement and an *exception*. The back edge from *S22* to the region vertex *R1* depicts the explicit flow to the beginning of the while-loop.

- **Dependence Edges**
  These are uni-directional edges that show control dependence between the vertices in the representation. Consider the example in figure 4. The edge from *R2* to *S20*, signifies that execution of statement *S20* depends on control entering the region represented by *R2*. It should also be noted here that Control Dependence Edges always start from the Region, Predicate or Method/Procedure vertex.
- **Call Edges**
  These edges start from the call site but the end point depends on whether the call is to a free standing procedure or to a method. A *Call* edge to a free standing procedure ends at the *Procedure* header, whereas for a method, which is a member of a class, the edge terminates at the *Class* header. A more detailed description of *Call* edges is given in section 3.3.1.
  - *Simple Call*
    A *Simple Call* edge represents a call to a method or a free standing procedure. This edge represents a call that can be resolved statically and represents the flow of control and data between the call site and the method or procedure called.
  - *Instantiation*
    Instantiation is the creation of an instance of a class. This is done by a call to the constructor of the class that initializes the object state and brings it into the system[3]. A special edge is used to represent the importance of an instantiation call. The *Instantiation* edge connects the instantiation statement to the *Class* header whose instance is created.
  - *Polymorphic Call*
    A call is said to be dynamically bound if it is not possible to resolve the call to

[3]In C++, a call to the constructor is implicit and is done at the point of instantiation of an object. In Smalltalk, the call is done explicitly.

a particular method at compile time. The *Polymorphic Call* edge is used in the representation of polymorphism and dynamic binding and is discussed in more detail in section 3.3.3. Briefly, the *Polymorphic Call* edge is an edge between the call site and the *Class* header of the class to which the variable is statically typed. The actual call may be to the class, indicated by the static type, or any of its subclasses.

- **Polymorphic Choice Edge**
  A *Polymorphic Choice* edge connects two members in the set of possible implementations to which a polymorphic reference can be resolved at run-time. This edge starts at a *Virtual Method* header (see section 3.2.1) in a superclass and terminates at the header for another virtual method with the same specification in the subclass. A polymorphic reference to a superclass virtual method, can be resolved to any virtual method in the hierarchy from that level downward. The *Polymorphic Choice* edge plays an important role in depicting polymorphism in the OPDG. Whenever a polymorphic reference is encountered, the reference is resolved by traversing the *Polymorphic Choice* edges that connect the candidates for resolving that reference.

- **Membership Edges**
  The concept of an entity being a member of some other bigger entity is used to group similar parts of the representation. Grouping of such similar entities allows easier understanding of the representation.

  - **Inheritance**
    The concept of inheritance is a central concept in object-oriented design. The *Inheritance* edge signifies membership of classes in an inheritance hierarchy and connects a subclass to it's superclass(es), in the direction of dependency.
  - **Class Membership**
    Every class definition has a fixed number of methods associated with it. As such, each method, including the constructors and destructors of a class, belongs to that class and is addressable only through an instance of that class (or the class itself in case of some object-oriented languages). A *Class Membership* edge connects the *Method* header and the *Class* header of the class in which the method is *defined*.
  - **Inherited Method**
    A method is said to be *inherited* if it is defined in a superclass and is inherited into a subclass. The inherited method is just like any other method in the subclass, and is also addressable in a similar way. Thus, an inherited method can be considered to have an implied membership in a subclass. The representation highlights this aspect, and the *Inherited Method* edge connects the subclass header to the header of the inherited method.

    One of our goals is a compact representation; however, here is a good example of a trade-off for more efficient algorithms. The edge back to the definition of an inherited method could be omitted and deduced when needed. Inorder to do this, the algorithm would first search the list of method headers in the local class and then follow the inheritance edge to the parent class, search that method list and continue up the inheritance hierarchy until the correct method is located. A method name may be overloaded; therefore, the algorithm must pattern match the complete signature to determine that it has found the appropriate method. In our view, the space required for the pointer is worth the resulting runtime efficiency.

3.3 Discussion

This section discusses the definitions given in the previous section. We concentrate on the important edges and vertices defined for an OPDG that make our representation suitable for object-oriented programs. Major issues presented the representation of polymorphism and dynamic binding, the inheritance hierarchy and a modified representation for handling parameters and parameter flow.

3.3.1 Call Edges

The *Call* edge to a method terminates at the *Class* header rather than the *Method* header. We use this approach to represent dynamic binding and to resolve issues of inherited method calls. The header of a class is equipped with information regarding its member functions, inherited methods, its place in the hierarchy, and its data attributes. We include in the OPDG, all static information necessary to resolve run-time issues about dynamic binding of a call to a method. Consider the following declaration statements in C++:

```
C *c1;  // C is a class in a hierarchy
C c2;   // c1 and c2 are instance variables
```

Both *c1* and *c2* have a static type of class *C*. However, *c1* can have a different dynamic type [17], if there are subclasses inheriting from *C*, since *c1* can be made to point to an instance of C or any of its subclasses. Whenever a message is sent to an object whose static type is the class *C*, the *Call* edge terminates at the *Class* header for *C*. Thus, in our representation, a *Call* edge representing a call to a method, terminates at the header of the class that dictates the static type of the object.

Consider the case of *c1* above. A message to an object pointed to by *c1* can be dynamically bound provided dynamic binding to the method is allowed by the method's declaration[4]. If the call is dynamically bound, the call is traced down to the actual implementation of a method to execute using the *Polymorphic Choice* edge and thus the dynamic reference is resolved.

If the message is to an inherited method, it can be traced using the *Inherited Method* edge from the *Class* header to the header for the inherited method. If the *Call* edge to an inherited method terminates at the *Method* headers, then tracking the definition site for the method would be a problem. Although the object is an instance of a subclass, the *Call* edge would be pointing to a method defined in a superclass. By making the *Call* edge terminate at the *Class* header, we eliminate the additional clutter due to the connections between call sites and headers for inherited methods.

Thus, inherited method calls and polymorphic calls are now processed in a manner similar to a call to a method that is defined in the class. The uniformity with which a message to a method can be traced makes this representation compact[5].

3.3.2 Parameter Handling

Object-oriented systems exchange a number of messages between their constituent objects. These messages pass control to the objects, and request operations to be performed on the data attributes within the object. The scoping rules prevent direct public handling of data and hence any operation on the data can be done only through a method in the interface of the object.

[4] In C++, this means that the method is virtual.

[5] Recall that a call to a free standing procedure is also represented by a *Call* edge, but this edge terminates at the *Procedure* header, since the procedure is not bound to any class.

Messages in object-oriented programs contain parameters to exchange information and hence handling of parameters is another problem we address in this paper. The traditional technique of handling parameter flow is to have additional edges between the call site and the called procedure, connecting the actual and corresponding formal parameters [12][19][28]. Since a number of messages are exchanged in an object-oriented system, this technique introduces excessive edges into the representation. Each message includes the *Call* edge, and along with it, at least one edge for each parameter (two for a pass-by-reference parameter).

The reason for these additional edges is to depict the flow of information between the actual and formal parameters. We expand the meaning of a *Call* edge to represent parameter flow in both directions, namely, from the call site to the method or procedure, and vice-versa. This additional meaning to the *Call* edge eliminates the edges between the actual and formal parameters thereby leading to a clearer representation.

The in-parameters are represented by *Parameter* vertices at the call site and the method or procedure header. Parameters that may-be-modified are each represented by a *Parameter* vertex at the extreme right of the call site and the *Method* or *Procedure* header. A one-to-one mapping is defined between the *Parameter* vertices at the actual and the formal sites. Thus, edges between the actual and formal *Parameter* vertices are obviated, making the representation more straight forward and easier to understand.

3.3.3 Representing Polymorphism

A polymorphic reference in an object-oriented language can, over time, refer to instances of more than one class [17]. Thus a polymorphic reference has both a static and a dynamic type associated with it. Polymorphism is associated with dynamic binding since the binding of a polymorphic message to the code to be executed in response to the call is accomplished dynamically. This means that the code associated with a given procedure call is not known until the moment of the call at run-time. The resolution of the polymorphic reference at execution time is done based on the dynamic type of that reference. The representation reveals a static view to this dynamic feature of the object-oriented paradigm. Polymorphism is explicitly represented using the *Polymorphic Call* edge, the *Virtual Method* headers and the *Polymorphic Choice* edge.

A *Polymorphic Call* edge is a special edge that represents a call to a method (specifically, a virtual method in C++). Polymorphic references are so termed because the references may be to more than one class. As such, the call can be bound to an implementation for a method with the same specification in any of the classes *down* the hierarchy. In our static representation of the potential run-time environment, this feature is represented by attaching a list of virtual methods to the *Polymorphic Call* edge. Once the polymorphic reference is resolved at run-time, the actual reference is used to access a method in the list and this method gets executed.

The representation of virtual methods and the concept of polymorphism is illustrated with an example as shown in figure 5. The virtual method *void do_something()* is redefined in class B. The destructor of base class A is declared virtual (following safe class design principles). The destructor in class B is also virtual since the destructor of the base class of the hierarchy is virtual. Since polymorphic calls in C++ can only be to virtual methods, each virtual method in a class has a *Polymorphic Choice* edge to a method with the same specification at the next level in the hierarchy (edge between the *do_something* methods in classes A and B).

An exception to the signature rule given above is the *Polymorphic Choice* edge between destructors of classes in the hierarchy, provided a superclass destructor is virtual (edge between the methods $\sim A()$ and $\sim B()$ in the example). Virtual destructors of a base class allow dynamic resolution to the destructor of any of the subclasses in the hierarchy and not otherwise.

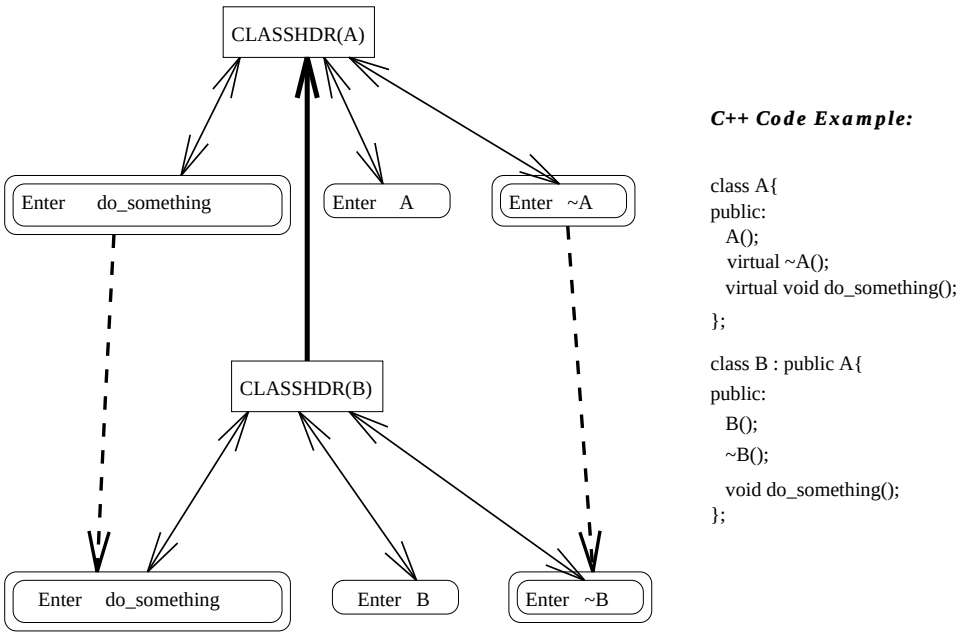


Figure 5: Representation of Virtual Methods and Polymorphism

Tracing the *Polymorphic Choice* edge from a *Virtual Method* header down the hierarchy will lead to the set of possible methods that can be dynamically bound to a polymorphic call. Our solution is to maintain information about the incremental growth of the set of possibilities for the polymorphic call. As the representation is built, the *Polymorphic Choice* edges are created between a virtual method in a class and its namesake method in its subclass. As classes are developed in a progression down the hierarchy, more choices are linked together.

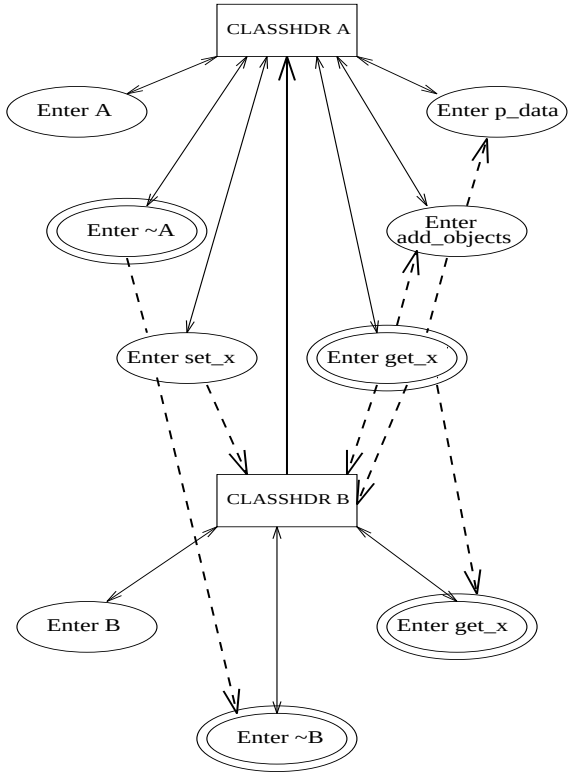
3.4 Class Hierarchy Layer

The **Class Hierarchy Subgraph (CHS)** is a graphical representation of the inheritance relationship between classes and the composition of methods into a class definition. Figure 6 shows the CHS for the example system in this paper. The code for the complete example is found in Figures 19, 20, and 21.

The CHS is built using the class specifications and may be built early in the design phase of a project. Method headers are created for each **new** definition of a method. That is, method header nodes are created for each method when it is defined the first time or when a method is overridden. A method header is attached to the header for the class in which it is defined. Subclasses that inherit the method have an edge that points to the latest method header for that method. This *reuse* of method headers results in a compact representation that faithfully represents the design of the object-oriented software.

The CHS does not contain any of the implementation details of the methods. Each method header can be thought of as belonging to both the CHS layer and the CDS/DDS layer. In the CDS/DDS layer, the method header is connected to a PDG for the code in the method. The details encapsulated within the method header, together with the corresponding PDG, represent the complete implementation of the method.

The system represented in Figure 6 contains five classes organized into three hierarchies. Class B inherits from class A, and class D inherits from class C. The double ovals indicate several dynamically bound methods: get_x in the classes A and B, get_x in the classes C and D, and the destructors A, B and C and D. Each of these pairs of methods are joined by polymorphic choice edges.




```
class A
{
private:
  int x;
protected:
  C C1;
  D D1;
public:
  A(int i=10);
  ~A() {};
  virtual int get_x();
  void set_x(in val=-1) ;
   int add_objects(C *amt);
   void p_data();
};


class B:public A
{
private:
  int x;
public:
  B(int i=100);
  ~B() {};
  virtual int get_x();
};
```

Figure 6: The Class Hierarchy Subgraph (CHS) for the example program

3.5 CDS and DDS Layer

3.5.1 Control Dependence

The control dependence subgraph (CDS) portion of the second layer of our OPDG is constructed in the same manner as the CDS described in section 2.2. Techniques for constructing the CDS have been presented in references [27, 22]. A CDS may be constructed for each method of a class in the CHS. All statements in the source program can be resolved statically.

The binding of polymorphic messages is resolved at compile time by the creation of a list containing polymorphically equivalent methods. These methods are connected to each other by polymorphic choice edges and to the call site by a polymorphic call edge. The polymorphic choice edges are directed since, for a specific class, only methods defined in the class' subclasses are polymorphically equivalent to the method defined in the class.

The CDS for a method is attached to the method header. The CDS is constructed once and then reused by each inheriting class that does not override the method. This results in a more

compact representation.

3.5.2 Representation of Parameters

The high number of messages exchanged in an object-oriented system, makes the issue of handling interprocedural control and data flow analysis an important one [18]. These messages pass control to the receiving objects and request operations to be performed on the attributes within the receiving object.

In object-oriented software, objects introduce a new level of scoping. The attributes of an object are visible within every method of the object. We have treated these attributes as global to each method of the object. The variables defined locally within a method are handled by standard intraprocedural techniques.

We modified the interprocedural representation of parameters to handle both explicit parameters and global variables, viewed as implicit parameters, in a homogeneous manner. Explicit parameters are handled by listing them in the vector in the same order in which they appear in the method signature. The existence of global variables in the system are handled as described in reference [6]. Globals are treated simply as pass-by-reference parameters and included in the vector as well. Hence, the method header for each method contains an n-element vector which represents any global variables, the attributes for the object that contains the method, and the parameters of the method, in that order.

When a subclass is created, it inherits from the base class(es) making those attributes accessible to the subclass in addition to its own attributes. In C++, if the inherited attributes are **private**, only inherited methods may modify the inherited attributes in the subclass, but the attributes are still part of the subclass. In fact, no object-oriented language, of which we are aware, allows for the actual removal of an attribute. As we progress down in the class hierarchy, the n-element vector grows monotonically larger due to the additional attributes defined at each level.

We use an incremental approach for constructing the n-element parameter vectors to reduce the effort required for construction. A vector is constructed for the class first and stored in the class header vertex. This vector includes any globals, the inherited attributes and any attributes defined locally in the class. The n-element vector for each method of a class is constructed by simply taking the generic vector for the class and adding in the formal parameters for the method currently being examined.

The values that variables may assume are objects and, in some languages, primitive values such as the integer, float, and char types of C++. To provide a representation that is language independent, we treat these primitive values as objects that belong to a class corresponding to their type. The operations defined on the types are treated as the methods of the class.

A data dependence edge from the vector entry to a node within the method indicates a variable's definition or use inside of the method. If the first occurence of the variable is a definition of the variable, the data dependence edge is an output dependence edge. Otherwise, the edge is a def-use data dependence edge to the use of the variable inside the method. The definition and use of an object as well as the technique for obtaining the data dependence edges is given in the following section.

3.5.3 Object Flow

Two important issues in interprocedural analyses are the linkage between definitions and uses of variables and the modification of the objects referred to by a variable. Variables are within scope in a block through one of several possible scenarios: (1) the variable is defined within the block;

(2) the variable is global to the block and no variable with the same identifier is defined within the block; or (3) the variable is passed as a parameter to the block.

We can observe the path an object follows from the entry in the n-element vector to its definition or use in a method. If the object is subsequently used as a parameter to a message, then we can follow its path to the next level's n-element vector of the corresponding method header.

> When an object is defined at one point in a program and is used or redefined at another point in a program, we say an *object flows* from the first point to the second.

Object flow is different from **data** flow because not only do attributes flow with an object from definitions to uses, but the methods that act on the attributes flow with the object as well. This is a fundamental feature of object-oriented software. Objects encapsulate *data* and *control* into a single entity [30]. Figure 7 shows the method header, *set_x*, of the example system with the DDS added.

The *object flow graph* for a system is a set of flow graphs some of which can be resolved statically and some of which can only be totally resolved dynamically. Where a flow cannot be uniquely resolved statically, the polymorphic choice edges that connect all possible polymorphic substitutes are used to construct a network of possible flow graphs.

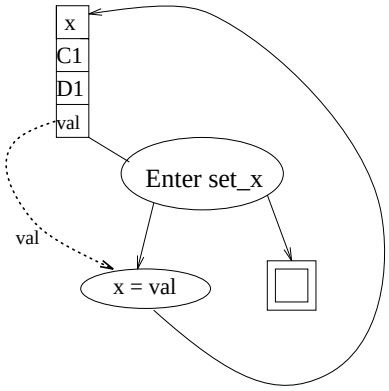


Figure 7: The CDS/DDS for method *set_x* of class A

3.5.4 Data Dependence

The DDS is constructed using the definition of object flow presented above. For the primitive values in a language such as C++, the object flow definition of a DDS corresponds to the usual definition. In a DDS, edges represent the dependencies among the program components, showing that the program's computation may be changed if the relative order of the nodes in the subgraph is changed. These dependencies run from the definition of an entity to its use.

The terms *def* and *use* in an object-oriented context refer to the *definition an object* and to the *use of an object*. A *def* of a variable in imperative languages is an instruction which assigns a value to the variable. A *use* of the variable is an instruction in which the variable is referenced [16]. In an object-oriented program, the value, i.e. the object referred to by a variable, can be the same object, but the object can be internally modified to have a different state.

Messaging is the technique whereby one object invokes the behavior of another object and may, or may not, result in the modification of attributes of the object. When any of the attributes are modified, the object itself is considered modified. Viewing an object in terms of its response to stimuli emphasizes the independence of objects, i.e., each object is an autonomous component of a program in execution [7]. The most common form of a stimulus is invoking a method. [6]

Definitions of *def* and *use* that take into consideration the composite nature of objects are given below:

> The *definition (def)* of an object in an object-oriented program occurs when an object is sent a modifier message.

> The *use* of an object in an object-oriented program occurs when an object is sent a read-only message or when the object appears as an actual parameter to a message.

For example, the class definition for a *list* will usually include *Add*, *Remove*, and *IsEmpty* methods. *Add* is a modifier method that adds an item to the list, *Remove* is a modifier method that removes an item from the list, and *IsEmpty* is a read-only method that returns **True** if the list is empty when the message is received. *Add*, *Remove* and the constructor for the list are examples of methods that cause a def of the object to occur. *IsEmpty* is a use of the object.

Definitions and uses of objects can be differentiated, using the OPDG representation, by examining the n-element vector associated with the corresponding method header. If the bit of the n-element parameter vector associated with an attribute is set, then the method may modify that attribute and there is a *def* of the object receiving the message. Otherwise, the object remains unchanged and there is a *use* of the receiving object.

The computation of intraprocedural object flow is modeled after the iterative computations of intraprocedural data flow chains, that are well known, to get the Data Dependence Subgraphs for each method of a class[32], [18]. Data dependence edges are added from each object definition to all uses reached by the corresponding definition.

The computation of interprocedural definition-use chains requires tracking the uses of globals and parameters that can be reached across a message from one object to another. This information for the inter-procedural definition-use chaining is incorporated into edges from entries in the n-element vector at each method header to corresponding definitions and uses in the method. No data dependence edges are added from the parameter vertices at call sites to the n-element vector at method headers due to the implicit translation of parameter vertices to entries in the vector. We add data dependence edges from the n-element vector at each method header to corresponding definitions and uses of the formal parameters inside the method. This ensures that we can observe the flow of an object into the method whether it is modified or used.

There are two classifications of object attributes: class and instance. A class attribute is shared by all instances of a class while an instance attribute is unique to a specific instance of the class. In C++, instance attributes are the default while attributes labeled as *static* are the class attributes. If the attribute is a class attribute then there is a class dependence edge from the attribute in the n-element vector at the method header back to the class header indicating the attribute belongs to

[6] There are two types of methods, modifier methods and read-only methods. A *modifier method* is a method that defines or changes any of the receiver's attributes. A *read-only method* does not modify the receiver's attributes.

the class. The different types of call edges keep track of the calling context for the computation of data flow.[7]

Figure 8 shows the DDS for the main program, see figure 21, of our example system. The second layer of the OPDG includes the CDS and DDS for each method of a class as well as the new representation of parameters. Figures 9, 10, 11, 12, and 13 show the second layer for each class in our example system.

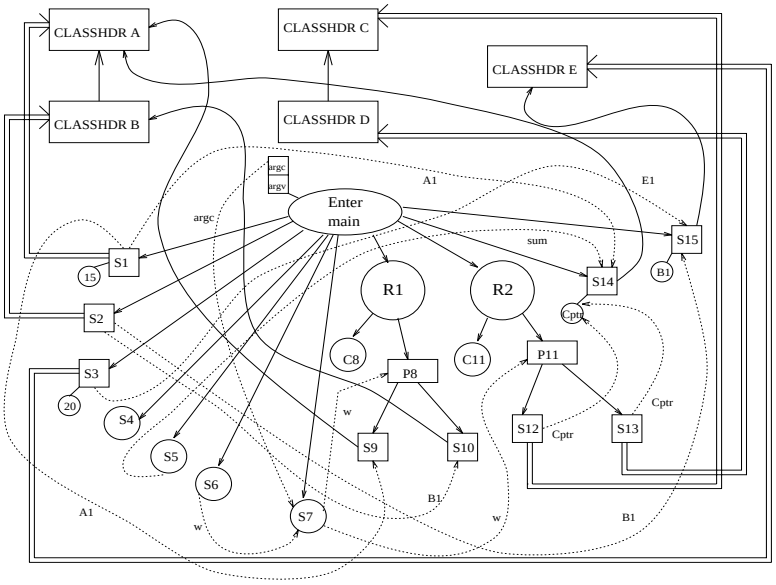


Figure 8: Complete representation for the main program

These first two layers of the OPDG represent object-oriented programs in a similar manner as the Unified Interprocedural Graph (UIG) represents procedure-oriented programs. The UIG provides control flow, data flow, control dependence, and data dependence sufficient for the computation of the information required for program maintenance. The same type of information contained in the UIG is represented in the OPDG, but not necessarily in the same way. The second layer of the OPDG contains control and data dependence edges, control flow edges, and call edges just as in the UIG. The information that the UIG's interprocedural flow edges and interprocedural reaching edges furnishes is provided via the type of call edge along with the n-element bit vector at each method header in the second layer of the OPDG. The modification of an implicit or explicit parameter is represented by toggling the corresponding bit of the vector. The parameter binding edges of the UIG are eliminated in the OPDG, but the translation information is represented in the left to right ordering of the parameter nodes. The features used in the UIG representing interprocedural information are also used in the OPDG.

3.6 Object Dependence Layer

The objective of the third layer is to provide a snapshot of the objects that are **live** at a specific point in a specific program execution. This layer of our representation is constructed from the

[7] In this paper, the techniques for computing the definition-use chains for object parameters and globals reside in an alias-free environment.

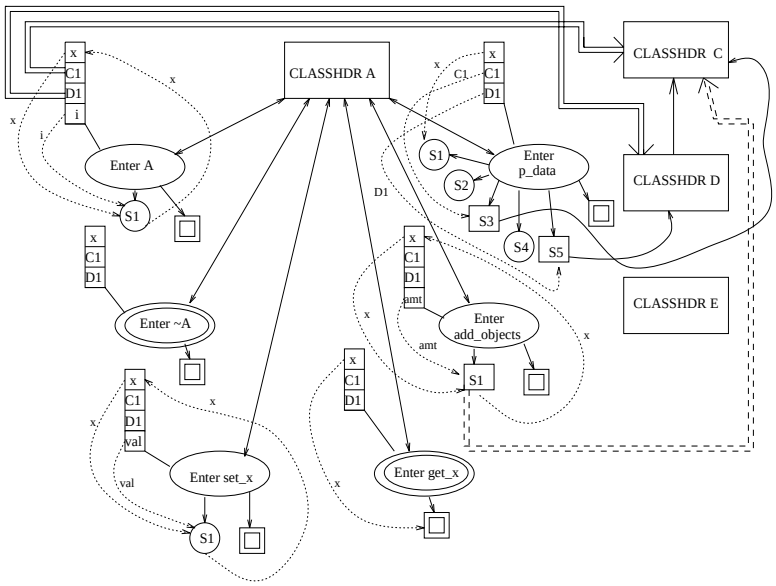


Figure 9: Complete representation for class A in the example class hierarchy

information presented in layers one and two and from inputs that identify a specific set of initial conditions. This layer supports the depth-first traversal of the composition hierarchy of a single object or a breadth-first traversal of peer-to-peer relationships with other objects at the same scoping level.

An object node represents information for a single object including an edge to the **Class Header** node for its associated class in the CHS layer and edges to the references to the object in the CDS/DDS layer. Two objects are connected by at most a single edge in the third layer, even if the objects have multiple associations with each other in layers one and two. Although the first two layers are required in order to compute this third layer; for many applications, the other two layers can be discarded once the third has been constructed.

3.6.1 Objects and Interactions

Objects are packages that contain data and the methods that manipulate them. While the second layer presents the details of these methods at a statement level, the third layer takes a higher level view and represents complete objects that contain several methods. Where the second layer provides details of interactions between individual statements and between methods, the third layer focuses on the interactions among objects and not the interactions between methods within an object.

Objects interact through sending messages, which in many ways are analogous to procedure calls in terms of the mechanism but not in terms of their intent. From a design perspective, messages establish a blend of control flow and data dependence. These two types of dependencies flow in opposite directions.

Intuitively, dependence means that the action of one entity directly affects another entity. Dependencies among objects are viewed as bidirectional because there are two equally important relationships subsumed within each messaging dependency. The direction of traversal of the dependence depends on whether control flow or data dependence information is needed for a particular

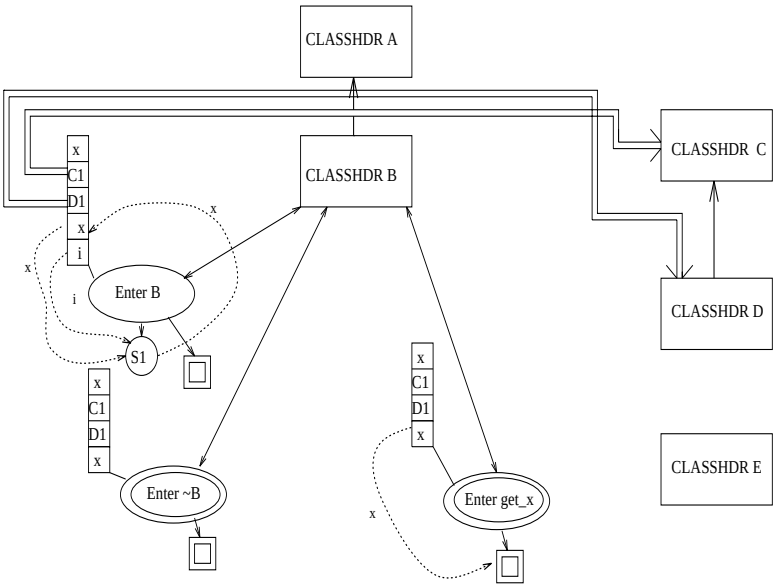


Figure 10: Complete representation for class B in the example class hierarchy

analysis. Two objects may have multiple control flow and data dependencies, but these are subsumed by a single object dependence. The exact cardinality of the dependencies is of no importance for many purposes. However, the cardinality is available in the second layer for those applications that require it.

Formally, we say an *object dependence* relation exists between **Object A** and **Object B** iff one of the following is true:

1. **Object A** sends a message to **Object B**, or
2. **Object A** receives a message with **Object B** as an actual parameter,
3. **Object A** is control dependent on a predicate that involves **Object B**, or
4. **Object A** and **Object B** share attributes (as in static or class attributes).

The first clause of the definition emphasizes the interaction of objects via messages. The *sending* object has a reference to the *receiving* object. Control flows from the sender to the receiver while there is a data dependence from the receiver to the sender.

For example, at the top level of a C++ program **main** is object dependent on any objects it messages. Similarly, within a method, the object containing the method is *object dependent* on any objects being sent messages inside the method. Conversely, the recipient of a message only acts when the sender messages and relinquishes control. The receiver is also object dependent on the sender of the message.

The receiver object may be defined within the code of the sender. For example, an object receives a message from the main object and the receiver is an attribute of main or an object that is local to a method is sent a message by the object that encapsulates the method.

The direction of object dependence flows from the receiver to the sender since the sender relinquishes control to the receiver by sending the message. The direction of object dependence

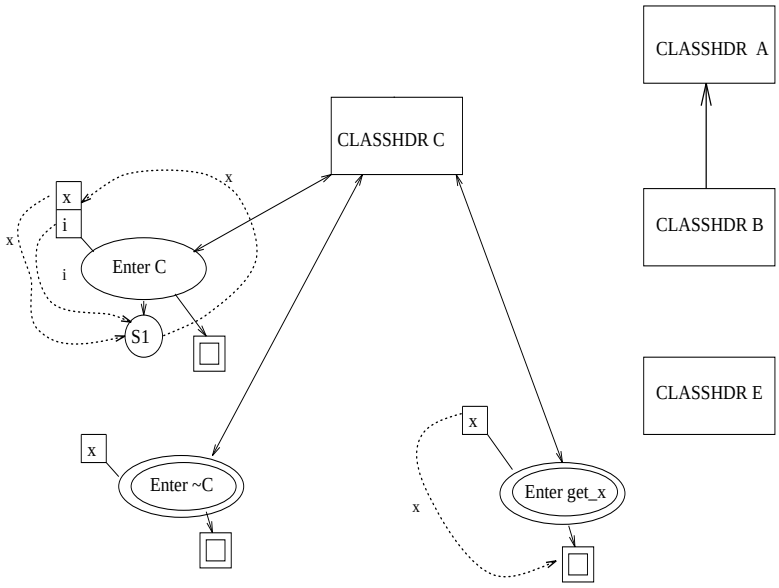


Figure 11: Complete representation for class C in the example class hierarchy

flows from the sender to the receiver since the state of the sender may change due to the receiver's actions. This is because the receiver is an attribute of the sender and when the state of the receiver changes, the state of the sender changes. Thus, the object dependence between and sender and receiver is bidirectional.

The second clause of the definition recognizes that objects sent as parameters of messages will, in turn, be sent messages by the receiver of the original message. The original recipient of the message is responsible for relinquishing control to the actual parameters. Thus, the actual parameters are object dependent on the object receiving the original message. This observation allows us to construct the object dependencies without descending into each method's implementation and applying the first clause multiple times. The following example illustrates this type of object dependence.
An object, O_R, receives a message, M_1, with two parameter(s), O_2 and O_3 from object O_S.

The direction of object dependence flows from O_2 and O_3 to O_R since the receiver relinquishes control to the parameters when it messages them. The direction of object dependence flows from O_R to O_2 and O_3 since the state of the receiver may change due to changes in the state of the parameter(s). Thus, the object dependence between receiver and parameter(s) is bidirectional.

There is the potential for multiple messages to be present in a single statement which will be represented by a single statement vertex. In this case, temporary objects are used to pipe the results of one message into another. Since the introduction of temporary objects is compiler specific, we fold an object dependence on a temporary object into the last receiving object in the sequence. This simplifies the representation of the third layer.

The third clause of the definition states that object dependence can be due to control dependence. Control dependencies can be represented as messages in most object-oriented code. A control structure determines the order of some activities. The fundamental control structure provides that a sequence of expressions will be evaluated sequentially. For example, the vertices, **S1**,

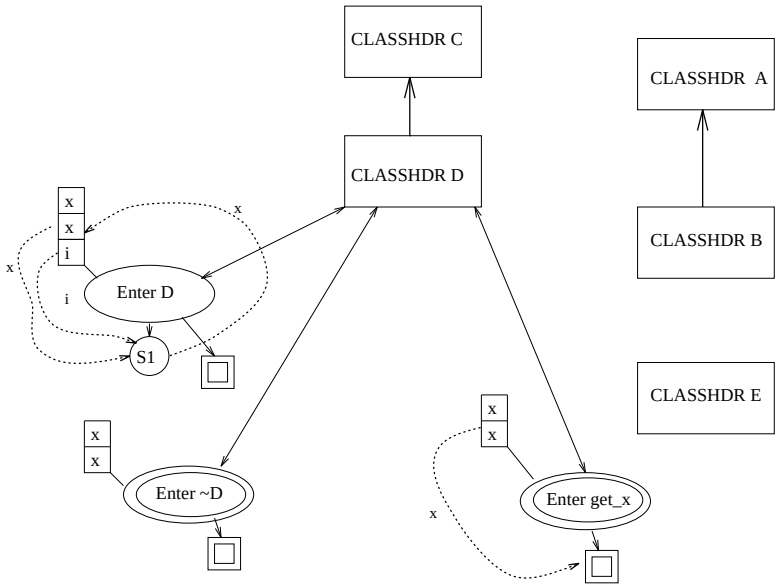


Figure 12: Complete representation for class D in the example class hierarchy

S2, and **S3** are control dependent on **main** in our example system in Figure 21. Also, there are implicit control dependence edges between the attributes and the instance of a class when it receives an instantiation message. We use this convention since the attributes are contained in the n-element vector associated with each method header of a class, but only are only sent instantiation messages when the constructor of a class is invoked.

This portion of the definition primarily addresses nonsequential control structures. In languages such as C++, the receiver of a message in the predicate part of a **if/then/else** control structure affect the state of the sender. The state of the sender then determines which objects receive messages in the **then** and **else** part of the same control structure. We map this to object dependence. The object dependence is bidirectional as described in the following case:

Languages such as C++ employ control structures that are represented by explicit control dependence edges in a CDS. These edges are mapped to a bidirectional object dependence.

The direction of object dependence flows from the object receiving a message in a statement that follows the predicate evaluation to the sender due to the control dependency between them. The control dependency is due to the fact that the state of the sender determines whether or not the receiver object may act. The direction of object dependence also flows from the sender to the receiver since the sender's new state has a data dependency on the receiver because the sender is affected by the receiver's actions. Thus the object dependence is bidirectional.

The fourth clause of the definition indicates that the existence of shared attributes leads to an object dependency among the objects sharing the attributes. These objects interact implicitly through their common attributes (static attributes in C++). If two instantiations of a class exist simultaneously, i.e., A_1 and A_2 are instantiations of class A, then there are data dependencies between these objects if there exists one or more static attributes. There will be a def-use flow dependence if, by messaging, one instance of the class defines the common attribute and the other instance uses it. If both instances modify the common attribute, then an output dependence exists. Hence, there is an object dependence between the two instances of the same class. The following

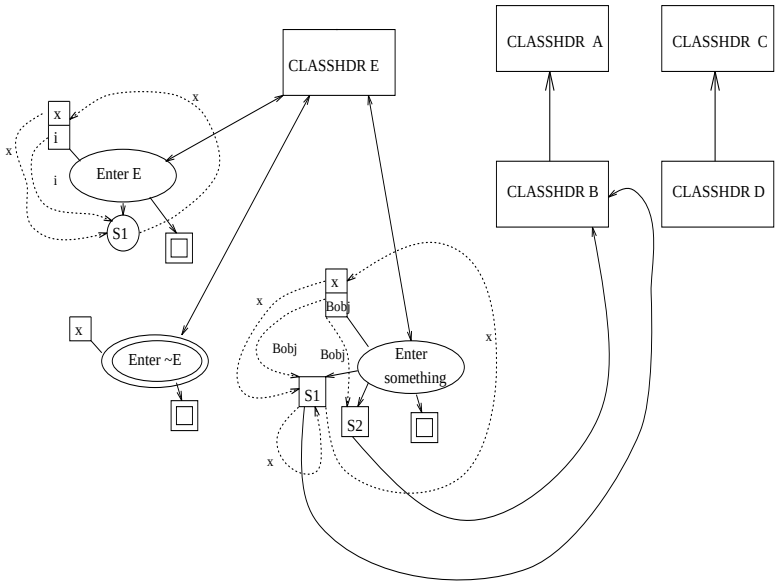


Figure 13: Complete representation for class E in the example class hierarchy

examples illustrate this type of object dependence.

Two instances, O_1 and O_2, of a class, C_1, share an attribute defined in the class.

If either instance modifies the shared attribute and the other uses or modifies the attribute there exists a data dependence. This dependence may occur in either direction depending on which instance accesses the attribute first. Hence, the object dependence is bidirectional.

3.6.2 Example of Object Dependence

Information about the interactions of objects is obtained by applying the definition of object dependence to each message contained in a node of the second layer of the representation. From our example program given in Figures 19, 20, 21, C_1 is A_1's class attribute. Hence, the constructor of class C will be invoked due to the instantiation of A_1. The object **main** sends an instantiation message that creates A_1 and then A_1 sends an instantiation message that creates C_1. There is a bidirectional object dependence between A_1 and **main** as well as between C_1 and A_1.

A similar situation holds if A_1 sends a message to objects other than to its attributes or parameters to the method, i.e., object C_1 is created in A_1's method and destroyed when the method terminates. When A_1 receives such a message it sends an instantiation message to create C_1. Alternatively, if C_1 is an actual parameter to one of A_1**'s** methods, then A_1 sends a message to C_1, i.e., a *use* or *def* of C_1 inside the method. In both cases there also exists a bidirectional object dependence between C_1 and A_1.

There is also the possibility of object dependence between an object and itself. The statement

$$\mathbf{x = x + 1}$$

contains two messages. There exists a bidirectional object dependence between the parameter, **1**, and the receiver, **x**, in the first message, "+". Since the *value* of the first message is used as a

parameter to the second message, "=", **x** is object dependent on itself. To avoid self loops, object dependence edges for objects dependent on themselves are not represented.

If calls to methods are bound at compile time, the object dependencies can be computed statically. In his work on modeling interactions, Wegner concludes that one "can't fully describe an object-oriented program statically." Thus, the interaction of objects occurs at both compile time and runtime.

3.6.3 The Object Dependence Subgraph

An **Object Dependence Subgraph (ODS)** depicts the object dependence relationships present in the system. The nodes of the ODS represent objects in the program. The edges represent the bidirectional object dependencies, subsuming both control flow and data dependence. We construct the third layer, the Object Dependence Subgraph, using a *projection* of the second layer. By projection, we mean that all the edge information represented in the second layer maps to some object dependence edge in the ODS depicting object interactions. Since we are collecting attributes and methods into objects in the third layer, the projections from the second layer may extend to several instantiations of objects from one class. We use the ODS to visualize the interaction of objects without all the extraneous information of the first two layers of the OPDG.

We map all types of edges from the second layer to the third layer to show how the actual objects of the program interact dynamically for a particular program execution. An edge in the ODS may correspond either directly to an edge in the second layer, represent multiple edges, or an edge from the second layer may be contained within an object and not explicitly represented in the third layer. Even those edges contained within an object are "mapped" so that algorithms can retrieve information from the second layer by traversing the inverse mapping. The bidirectional object dependence edges of the ODS represent inter-class edges.

We provide an itemized mapping for the edges of the second layer to define the projection of the second to the third layer. The mapping is as follows:

- An *explicit flow edge* maps directly to an object dependence edge. Since this is actually a control dependence edge depicting a loop back edge in a program, it is mapped to the part of the object dependence edge that flows in the same direction as the control flow.
- A *data dependence edge* is mapped to the null set and is not explicitly represented in the third layer since both ends of the edge are within a single object.
- A *static data dependence edge* is mapped to bidirectional object dependence edges between all instances of a class that share the static attribute.
- A *control dependence edge* maps to the null set and is not explicitly represented since the control dependence is intra-procedural and; therefore, is contained totally within one object.
- A *simple call edge* indicates a message. It maps to multiple object dependence edges. The first is the part of the bidirectional object dependence edge from the receiver to the original sender.

  An edge from the call site to a parameter vertex maps directly to an object dependence edge. It is between the actual parameter and the receiver of the message. For user defined messages, the parameter edge as well as the parameter vertex at the call site are explicit. The edge and vertex are implicit for non-user defined messages.

- An *instantiation edge* indicates a message to the class. It maps directly to an object dependence edge. The edge is a bidirectional object dependence edge from the instance of the class created to the object sending the message. Other bidirectional object dependence edge(s) between any actual parameter(s) of the instantiation message and the object instance created if the constructor contains any parameters.

- A *polymorphic call edge* indicates a dynamically resolved message and maps directly to an object dependence edge. The bidirectional object dependence edge is from the receiver to the original sender. Other bidirectional object dependence edge(s) between any actual parameter(s) of the instantiation message and the object instance created if any parameters exist as indicated above.

- A *polymorphic choice* edge is used with the polymorphic call edge to resolve the method to invoke by linking similar method definitions in the class hierarchy. It is mapped to the same object dependence edge as the polymorphic call edge.

- A *class membership edge* is followed during runtime after a call edge to the invoked method. Therefore, the class membership edge is mapped to the same object dependence edge as the corresponding call edge.

- An *inherited method edge* is followed during runtime from the class header to the invoked method residing at a higher level in the class hierarchy after following a call edge. The inherited method edge is mapped to the same object dependence edge as the corresponding call edge.

3.6.4 Construction of the ODS

This section provides the pseudocode algorithm for constructing the Object Dependence Subgraph. An outline of the OPDG class definition is provided. The methods, *ConstructPiece* and *HandleStatic*, are defined below. Since the meanings of the other methods are straightforward, the implementation details are not provided. The algorithm assumes an understanding of the discussion of the syntax and concepts pertaining to the first two layers of the OPDG. The construction process is performed recursively to capture the object interactions at all levels of object encapsulation until the ODS is completed. An algorithm to construct the ODS is presented in Figure 14.

3.6.5 An Example of Constructing the ODS

An ODS for a program is constructed for a given program input, and a given point in the execution. The ODS represents the objects that are live at the selected point in the execution. Figure 15 shows the ODS resulting from applying the algorithm discussed in the previous section for the example system with **argc** = 26. The ODS depicts the object interactions that occur up through S8. We project the edges of the second layer of our representation to edges of the ODS by applying the definition of object dependence given in section 3.6.1 and the mapping given in section 3.6.3.

We will discuss constructing the ODS for our example system by following one thread of object encapsulation that begins with **main** and contains A_1 and its attributes. This will illustrate the different mappings the algorithm handles during the construction process. We will also discuss a

```
Input : The second layer of the OPDG containing the CDS and DDS for a program.
A specific set of input data and a specific point in the execution.
 Output: The Object Dependence Subgraph for a particular program execution.



class ODS-         // Class ODS definition
public:
   ODS(Graph OPDG);
   ~ODS();
   void Display();
protected:
   void ConstructODS(Graph OPDG);
   virtual void AddVertex(OPDGVertex vertex);
   void ConstructPiece(OPDGVertex vertex);
   virtual void StaticDependence(OPDGVertex vertex);
   virtual void HandleStatic();
   virtual void AddDependenceEdge(OPDGVertex vertex1, OPDGVertex vertex2);
private:
   Set objset;
";

ODS::ConstructODS(Graph OPDG):   // Main routine
   // The set of objects containing static attributes
   objset = EMPTY;
   // Add mainobject to ODS and label
   ODS.AddVertex(mainobject);
   ODS.ConstructPiece(mainobject);
   ODS.HandleStatic();
   ODS.Display();
"

ODS::HandleStatic() -//Handles object dependence between objects sharing attributes
   // Clause 4
   For each object in objset
     currentobject = object;
     For each object in objset
        If object /= currentobject and class(object) /= class(currentobject)
           AddDependenceEdge(object, currentobject);
        EndIf;
     EndFor;
"

ODS::ConstructPiece(OPDGVertex vertex):
   While there are control dependence edges to follow from vertex
      // Clause 1 if following a sequential control dependence edge
      currentedge = nextedge;
      currentvertex = nextvertex;
      For each message in currentvertex
         // Clause 1
         ODS.AddDependenceEdge(receiver, sender);
         // Clause 2
         For each parameter to message
            If not a reference parameter
               ODS.AddVertex(parameter);
            EndIf;
            ODS.AddDependenceEdge(parameter, receiver);
         EndFor;
         If StaticDependence(vertex)
            objset.Add(vertex);
         EndIf;
      EndFor;
      // Recurse for subsequent levels of encapsulation
      While unvisited callsite vertices
         Mark callsite visited;
         Follow call edge to class header;
         If polymorphic call
            Follow membership edge to method header;
            While polymorphic choice edge exists
               Follow polymorphic choice edge to next method header;
            EndWhile;
         Else
           Follow membership edge to method header;
         EndIf;
         ODS.ConstructPiece(method);
      EndWhile;
   EndWhile;
"
```

Figure 14: Algorithm for Constructing the ODS

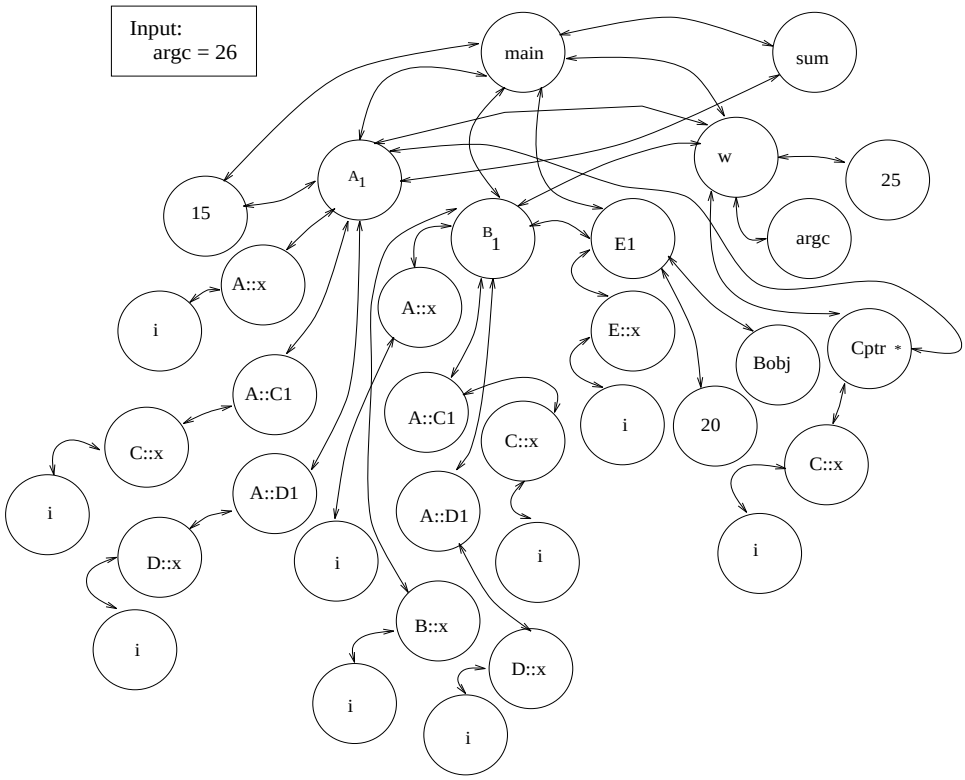


Figure 15: The Object Dependence Subgraph

few representative edges of the ODS, allowing the reader to construct the entire ODS step by step according to the algorithm provided in the previous section.[8]

The construction begins by adding **main** to the ODS.[9] The control flow edge emanating from **main** to the object A_1 maps to a bidirectional object dependence edge between A_1 and **main**, {1}. The instantiation message in **S1** 21 contains a parameter. An object vertex and object dependence edge is added to the ODS to depict the object dependency between **main** and the parameter,{2}. The instantiation and class membership edges are mapped to a single object dependence edge,{1}. The edges are mapped to the object dependence edge representing the control flow edge between **main** and A_1.

Within the constructor of Class A, the class attributes of A_1 receive instantiation messages. We add object vertices and object dependence edges to the ODS for each of the attributes of A_1,{1}. The attribute, **x**, receives a modifier message within the the constructor implementation. There is bidirectional object dependence between the actual parameter **i** and **x**, and the corresponding object vertices and object dependence edge are added to the ODS,{2}. The control dependence edges from **main** to the objects, B_1, E_1, **w**, and **sum**, as well as the instantiation and subsequent edges encountered in the construction of these objects map to object dependence edges in the same manner as illustrated with A_1.

[8] As the naming convention, boldface words represent the objects referenced in the following discussion of the ODS. References to a specific clause in the definition of object dependency are enclosed in { }.

[9] Those languages that do not have an explicit **main** do have an object that initiates the execution. This is the first object added to the ODS.

The messages in statements, **S8**, **S10**, **S12** and **S16** map to multiple object dependence edges,{2}. There is a bidirectional object dependence between each object receiving a message from **main**. In addition, there are bidirectional object dependencies between the parameters to each message and the corresponding receiver. The object vertices and object dependence edges are added to the ODS. As we follow the call edges for each object's message, we discover more object interactions to represent in the ODS. We continue to descend further down into the implementation levels of each object's message, collecting all the object dependencies until there are no more call edges to follow.

We map the control dependencies between objects in statements **S9**, **S10** and **S11**,{3}. There are bidirectional object dependencies between both A_1 and **w** and B_1 and **w** due to the predicate (w ¿ 25). Hence, we add object vertices and object dependence edges to the ODS as the projection. Again in **S13**, **S14** and **S14** we have control dependence depicted in the second layer of the OPDG. The object vertices and object dependence edge are added to the ODS to reflect the interaction of the object that **Cptr** references and **w**.

We have illustrated the construction of the ODS corresponding to specific inputs and a specific point in the execution by explaining how to map some representative edges from the second to third layer of the OPDG. Subsequent application of the definition of object dependence gives the complete ODS as seen in Figure15. A large number of different graphs are possible by selecting a different point in the execution at which to stop or by selecting different initial conditions. Each edge of the second layer is projected to an object dependence edge in the third layer. The development of the ODS completes the comprehensive program representation, providing both static and dynamic information about an object-oriented program.

4 Implementation of the Representation

A prototype for the OPDG has been constructed. The architecture of the prototype models the three layers presented in this paper. Different tools will require different layers of the representation so the construction of each layer is a well-defined subsystem. The architecture also recognizes that the information required to build each layer is often available long before the information required for the next layer in the model.

The structure of the prototype is illustrated in Figure 16. The program code is parsed into an abstract syntax tree (AST). The AST is used by the OPDG interface layer which constructs the OPDG structure. The structure is created by first creating objects for the appropriate nodes in the AST. These objects are dependent on the language and even dependent on the parser itself.

Each subgraph in the OPDG has a manager that is reponsible for its construction. A manager object creates OPDG nodes and encapsulates within them the appropriate AST nodes. This wrapping of the AST nodes associates a language-specific construct with a paradigm-level concept. The OPDG structure is language independent in that each node represents a basic object-oriented concept. The structure of the OPDG is language dependent in that each language tends to promote the use of certain constructs over others and presents a pattern of use that is unique to that language.

The CHSManager traverses the AST, selecting the nodes needed to construct the CHS. At the appropriate points, the CHSManager may elect to notify the CDSManager and DDS Manager objects to construct their graphs. The CHSManager may be configured not to call these objects for those tools that only require the CHS. Likewise, the DDSManager can be configured to either notify or not notify the ODSManager.

The product of this construction process is a traversible graph which can be utilized by software

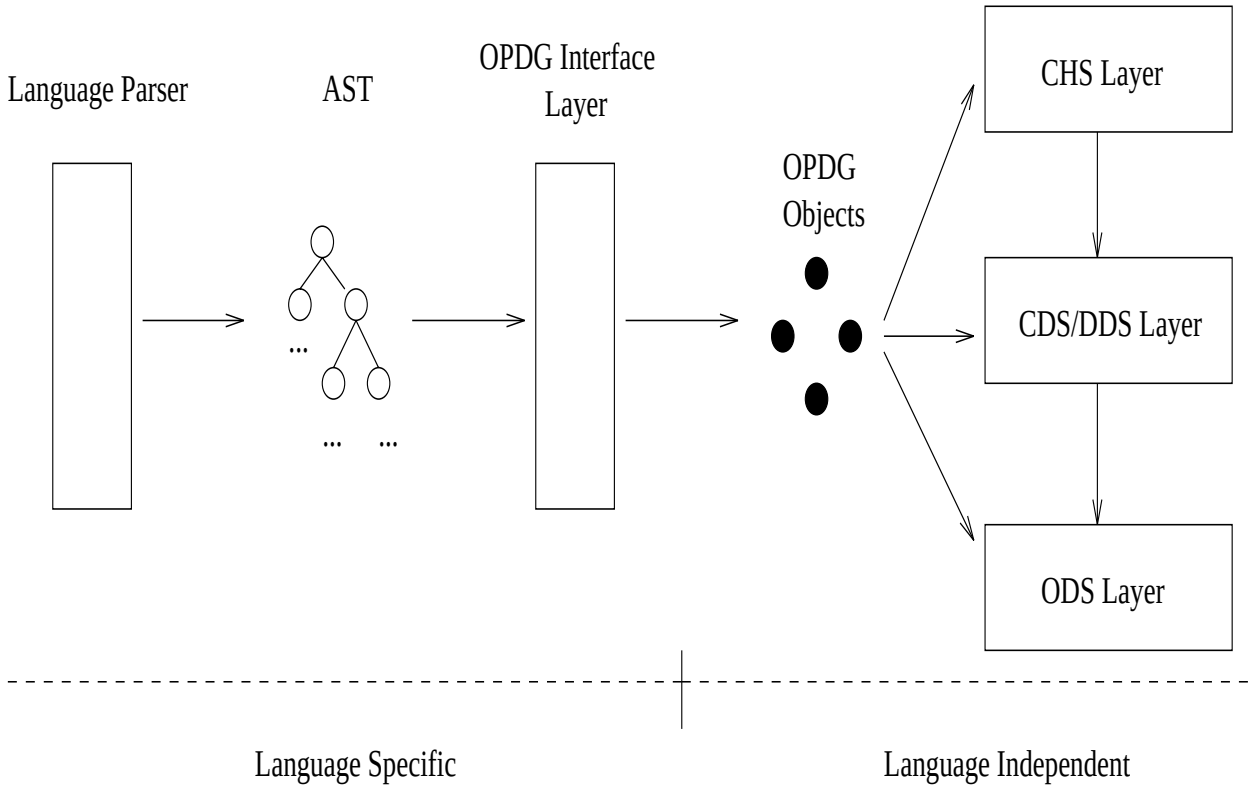


Figure 16: Architecture for OPDG Prototype

engineering tools. The three layers are interconnected. The method header nodes of the first layer are connected to the CDS/DDS graphs of the second layer by the connections via the parameter vector. The second layer is connected to the structures of the third layer by the projection edges used to construct the object dependencies.

Tool construction has begun on top of the basic representation. One of the tools that has been implemented calculates the Depth of Inheritance Tree (DIT) metric[8]. In this case, the metric was implemented in the CHSManager since the CHS has sufficient information to calculate the metric value. The DIT tool was implemented in a straightforward manner as a method of the CHS.

The classes that implement the OPDG constitute a framework for tool development. The tool developer configures the classes to select those portions of the representation to create. The tool developer selects the appropriate AST classes for the language/parser of their choice. The classes of the framework are then extended to implement the logic needed for specific tools.

The prototyping effort demonstrates that the representation meets its goals. The representation is compact since method information is stored only for its original definition and not repeated at inheritance sites. It is language independent but preserves the pattern of usage characteristic of a particular language. The representation provides a basis upon which tools can easily be implemented.

5 Applications of the Representation

To illustrate the use of the OPDG we will present two applications of the representation. The first application produces a slice of the ODS that provides support for building debuggers and other runtime analysis tools. The second application is the implementation of a simple class metric.

5.1 Layered Slicing

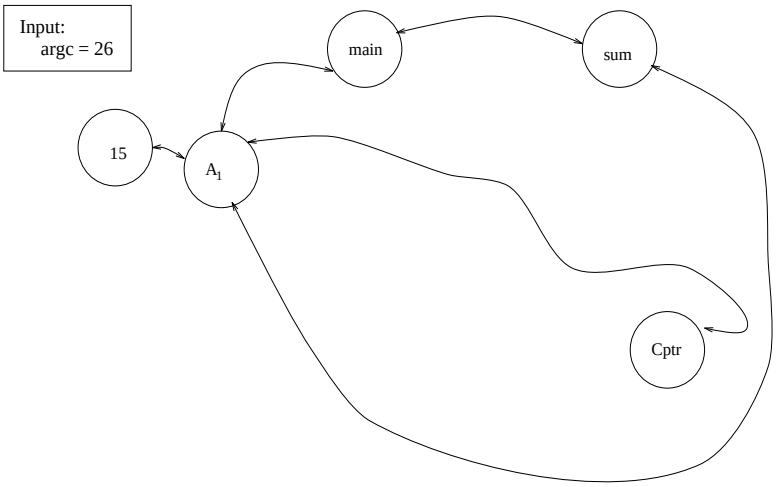


Figure 17: A horizontal slice of the ODS for the example program.

In this section, we introduce the concept of computing an execution slice of an object-oriented program using the OPDG. An execution slice was introduced by Weiser where a slice of a program is taken with respect to a program point p and a variable x. The traditional execution slice consists of those statements in a program that might affect the value of x at program point p[33].

For object-oriented software, we are interested in the program objects and their dynamic interactions at a particular point in the program. To compute a slice of an object-oriented program, we view the third layer of the OPDG, the Object Dependence Graph (ODS), as also being layered where the layers consist of different granularities of object encapsulation that are captured by a *horizontal slice* or a *vertical slice*. A *horizontal slice* with respect to a program point p and an object *obj* is a subset of the ODS that includes *obj* and those objects that are defined outside of *obj* that are live at p and are object dependent on *obj*. A *vertical slice* with respect to a program point p and an object *obj* is a subset of the ODS that includes *obj* and those objects that are live within *obj* at point p.

To illustrate a horizontal slice, consider Figure 17, where we show a horizontal slice of the ODS for the example program with respect to the point in the program immediately following execution of S8 and object A_1. The slice includes A_1, main, sum, B::x, i, A::D1, A::C1, A::x, and 15 together with the edges indicating the interdependencies between the objects.

To illustrate a vertical slice, consider Figure 18, where we show a vertical slice of the ODS for the example program with respect to the point in the program immediately following execution of S8 and object A_1. The slice includes A_1, A::x and parameter i, A::C1 and A::D1, together with the edges indicating the interdependencies between the objects. A vertical slice with respect to object A_1 includes only those objects that are live within A_1 at a given point in the program. To further illustrate the concept of a vertical slice, consider the execution sequence beginning at S14 in *main* where *main* is sending message *add_objects* to A_1 passing a pointer to a dynamically bound object. Execution continues in method *add_objects* of object A_1; consider the point after x is defined in *add_objects* and the slice of object A_1 at that point in the program. Figure 18 illustrates the vertical slice of A_1 at the point after x is defined where we include the dependence between A::x and Cptr.

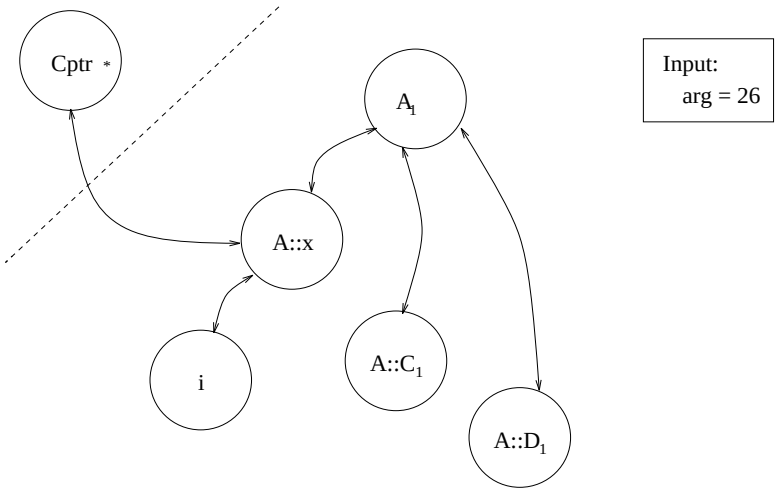


Figure 18: A vertical slice of the ODS for the example program.

5.2 Class Metrics

The OPDG contains sufficient information to support the calculation of many metrics about the code. Although metrics for object-oriented software are far from mature, there have been several efforts to describe a set of measures that characterize the products[8], [23],[20],[26]. In this section we will describe the use of the OPDG to support the development of metrics tools.

The technique for adding a new metric to the OPDG requires the following steps:

1. Identify a metric and its definition
2. Determine the scope of the metric
3. Use the scope to locate the appropriate object within the OPDG in which to place the required method
4. Translate the metric definition into an implementation using the structure of the encapsulating object

The Depth of Inheritance (DIT) metric[8] for a class and the average DIT for a design are intended to provide designers and managers with feedback about the quality of the design. A class's depth is measured by counting the number of layers of parent classes. The DIT for a class, in a single inheritance language, can be defined recursively as:

```
DIT(class) = if (parent == nil)
                    0
             else
                    1 + DIT(parent)
```

The information to calculate DIT resides in the CHS layer. A method can be added to the CHS object to traverse the CHS computing the DIT for each class and counting the number of classes in the subgraph. The average DIT is returned as the result of this method.

Each metric has some *scope* which in turn determines where the tool to compute that value should be located. The DIT metric focuses on a class and the method to calculate it could be located in the *class* node. In fact, since the value of DIT can not change dynamically, the value of DIT for a class could be computed at the time the class node is created and stored in that node. The average DIT for a design has the CHS as its scope and should be located in the CHS object.

The OPDG is supporting our concurrent research effort in object-oriented metrics. The structure of the OPDG provides a means of estimating the effort required to calculate a proposed new measure. The prototype OPDG tool allows us to quickly prototype the new measures and judge their effectiveness on a metrics test suite.

6 Conclusions and Future Work

The program representation, the OPDG, presented in this paper provides a detailed representation of object-oriented systems. The representation is divided into three layers. The first two provide the compile view and the third represents the runtime view of selected objects.

We introduced new definitions of *definition (def)* and *use* of variables for analyzing object-oriented programs. We adapted the concepts pertaining to data flow in the Program Dependence Graph and its variations to the new concept of *object flow*. We defined *object dependence* and its application in the dynamic component of our representation.

Algorithms were provided for constructing each of the layers of the OPDG. The layers are sufficiently modular to support the construction and use of a portion of the representation for all or part of an object-oriented program. The resulting structure is compact and easy to relate to the original program.

Finally, we discussed potential applications of our representation including a simple metrics tool and a program analysis technique termed layered slicing. Metrics at a variety of levels of detail can easily be calculated using the information captured in the representation. Applications take advantage of the modular structure to use only that part of the representation containing the information required.

An implementation of the OPDG has been built using the Smalltalk interactive environment and programming language. The implementation is a tool that can construct the OPDG at compile time for a C++ system. The tool receives the parser output of the compiler and builds the representation from it. Based on the output from the tool combined with system run-time information, another tool can construct the run-time picture of the system. Since the research provides a language independent representation, it is will be suited for applications in a variety of different tools.

Several representation issues, such as aliasing and recursion, require additional attention. Problems due to aliasing are similar to those seen in procedural systems, but are more complicated for object-oriented systems due to the existence of objects and polymorphism. The concepts of object flow and object dependence have not been explored sufficiently to discover all of their potential implications for the representation and analysis of object-oriented software.

```
#include<iostream.h>

class C
{
private:
  int x;
public:
  C(int i=15);
 virtual  ~C() {};
  virtual int get_x();
};

class D:public C
{
private:
  int x;
public:
  D(int i=25);
  virtual ~D() {};
  virtual int get_x();
};

class A
{
private:
  int x;
protected:
  C C1;
  D D1;
public:
  A(int i=10);
  ~A() {};
  virtual int get_x();
  void set_x(in val=-1) ;
   int add_objects(C *amt);
   void p_data();
};

class B:public A
{
private:
  int x;
public:
  B(int i=100);
  ~B() {};
  virtual int get_x();
};

class E
{
private:
  int x;
public:
E(int i=18);
  ~E() {};
  int get_x();
  void something(B Bobj);
};
```

Figure 19: The class header files

```
// Class C
C::C(int i) {
  x = i;
}

int C::get_x() {
  return x;
}


// Class D
D::D(int i) {
  x = i;
}

int D::get_x() {
  return x;
}

// Class A
A::A(int i) {
  x = i;
}

int A::get_x() {
  return x;
}

void A::set_x(int val) {
  x = val;
}

int A::add_objects(C *amt) {
  x = x + amt->get_x();
  return x;
}

void A::p_data() {
cout << "x = " << x << endl;
cout << "C1's data " << endl;
cout << C1.get_x() << endl;
cout << "D1's data " << endl;
cout << D1.get_x() << endl;
}

// Class B
B::B(int i) {
  x = i;
}

int B::get_x() {
  return x;
}


// Class E
E::E(int i) {
  x = i;
}

void E::something(B Bobj) {
  x = x + Bobj.get_x();
  Bobj.p_data();
}
int get_x(){
  return x;
}
```

Figure 20: The class definitions

```
main(int argc, char **argv) {
/* 1*/  A A1(15);
/* 2*/  B B1;
/* 3*/  E E1(20);
/* 4*/  C *Cptr;
/* 5*/  int sum;
/* 6*/  int w;

/* 7*/  w = argc;
/* 8*/  if (w > 25)
/* 9*/     A1.get_x();
          else
/*10*/  B1.get_x();

/*11*/  if (w > 25)
/*12*/     Cptr = new C;
           else
/*13*/     Cptr = new D;

/*14*/  sum  = A1.add_objects(Cptr);
/*15*/  E1.something(B1);
}
```

Figure 21: The main program

References

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language and Design and Implementation*, pages 20–22, 1990.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. *ACM Transactions on Programming Languages and Systems*, pages 384–396, 1993.

[3] W. Baxter and J. R. Bauer. The program dependence graph in vectorization. *Sixteenth ACM Principles of Programming Languages Symposium*, pages 1–11, January 1989.

[4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. *Proceedings of the Conference on Software Maintenance '92*, pages 41–50, 1992.

[5] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City,CA, 1991.

[6] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of SIGPLAN'88 Conf. Programming Language Design and Implementation*, pages 47–56, 1988.

[7] T. Cargill. *C++ Programming Style.* Addison-Wesley Publishing Company, Inc., Massechusetts, 1992.

[8] S.R. Chidamber and C. F. Kemerer. Towards a metric suite for object-oriented design. In *Proceedings of OOPSLA '91*, pages 197 – 211, 1991.

[9] E. Duesterwald; R. Gupta and M.L. Soffa. Rigorous data flow testing through output influences. *Proceedings of the 2nd Irvine Software Symposium (ISS'92)*, pages 131–145, 1992.

[10] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[11] R. Gupta and M. L. Soffa. Automatic generation of a compact test suite. *Proceedings of the Twelfth IFIP World Computer Congress*, 1992.

[12] M.J. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, 19(6):584–593, June 1993.

[13] M.J. Harrold and M.L. Soffa. Computation of interprocedural definition and use dependencies. *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*, pages 297–306, March 1990.

[14] R. Gupta; M.J. Harrold and M.L. Soffa. An approach to regression testing using slicing. *Proceedings of the Conference on Software Maintenance '92*, pages 299–308, 1992.

[15] B. Korel. The program dependence graph in static program testing. *Information Processing Letters*, 24:103–108, January 1987.

[16] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(6):155–163, 1988.

[17] Tim Korsen and John McGregor. Object-oriented software design: A tutorial. *Communications of the ACM*, 33(9):40–60, 1990.

[18] B. Malloy; J.D. McGregor; A. Krishnaswamy and M. Medikonda. An extensible program representation for object-oriented software. Technical report, Clemson University, 1994.

[19] P. E. Livadas and P. K. Roy. Program dependence analysis. In *IEEE Conference on Software Maintenance 1992*, pages 356–365, 1992.

[20] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[21] B. A. Malloy, R. Gupta, and M.L. Soffa. A shape matching approach for scheduling fine-grained parallelism. *Proceedings of MICRO-25, The 25th Annual International Symposium on Microarchitecture*, pages 131–135, December 1992.

[22] M.J. Harrold; B. Malloy and G. Rothermel. Efficient construction of program dependence graphs. *ACM International Symposium on Software Testing and Analysis*, 18(3):160–70, June 1993.

[23] J. D. McGregor and S. Kamath. A psychological complexity measure at the domain analysis level for an object-oriented system. Technical report, Dept. of Computer Science, Clemson University, 1995.

[24] J. D. McGregor and D. A. Sykes. *Object-Oriented Software Development: Engineering Software for Reuse*. Van Nostrand Reinhold, 1992.

[25] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. *Conference on Programming Language Design and Implementation*, pages 266–277, june 1994.

[26] N. Ojha and J. D. McGregor. Object- oriented metrics for early system characterization: A crc card-based approach. Technical Report TR94-107, Dept. of Computer Science, Clemson University, 1994.

[27] J. Ferrante; K. Ottenstein and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(10):319–331, July 1994.

[28] 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.

[29] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regressin test selection. *Proceedings of the Conference on Software Maintenance*, pages 358–367, Sept 1993.

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

[31] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–225, May 1979.

[32] A.V. Aho; R. Sethi and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

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