CHAPTER 6
CLASS ANNOTATED GRAPHS

6.1 REPRESENTATIONS OF OBJECT-ORIENTED PROGRAMS

In order to reason about the impact of changes applied to an evolving class, an abstract representation is required. Such an abstract representation is, in general, useful to perform any kind of static analysis on a given program. Static analysis (Vinter 1997) refers to the process of analyzing a given program source (without executing it) for the purpose of gathering useful information about the program. Such a static analysis is frequently used by several programming language environments Rangarajan (1997) gives one such example.

Researchers have proposed several representations of programs that enable efficient algorithms to be applied. Some of these are control flow graphs, data flow graphs, system dependence graphs, program dependence graphs, and class call graphs. More recently Harrold and Rothermel (1996) have proposed a coherent family of graph representations for object-oriented software.

6.2 A NEW GRAPHICAL REPRESENTATION

One of the limitations of the existing graphical formalisms is that they are language-independent. Although this language independence, in general, is an advantage, there are certain important pieces of information specific to a programming language that need to be captured occasionally. For instance, none of the existing formalisms can explicitly represent the notion of private derivation supported by C++, or a synchronized method in Java. A reason for this appears to be the way these formalisms have gradually evolved from the structured programming period. At the same time, it is unwise to come up with language-specific representation schemes since there will be too many of those and general purpose algorithms cannot easily be designed for them.
One way of introducing appropriate language-specific details in the graphical representation without making the representation scheme itself bound to a certain language is to use the notion of *annotation*, as in UML (Muller 1997, Larman 1998). The annotation can then be used by an algorithm that requires language-specific detail. Such an approach is well suited to reason about the regression test appropriateness based on atomic changes discussed in this thesis. A representation called *Class Annotated Graph (CAG)* is proposed in this thesis as an extension of existing formalisms for use with various object-oriented languages.

6.3 ELEMENTS OF CAG

Figure 6.1 depicts the different elements of the CAG. A graph representation of an object-oriented program that follows CAG approach will comprise elements solely from this list.

![Diagram of CAG elements](image-url)
6.3.1 Class header

Each class in CAG will start with a class header node that includes the class name. Members of the class are connected to the header by membership arcs.

6.3.2 Method header

Method definitions start with a method header node. Name of the method is shown inside the node. Associated with a method header are method argument list and return specification. Both of these are optional.

The method header is also used at the call point, the difference being, the actual parameters are bound at the place of call with an arc connecting the call site to its corresponding definition.

6.3.3 Method argument

This symbol is used to denote each argument to a method. This may be additionally annotated to indicate the type of the argument, parameter passing mode (by value, etc).

6.3.4 Method return value

Methods that return a value will have this indicator attached to the left of the method header. The string \texttt{ret$}$ inside this node is used to propagate values from within the method to the outside environment. Value returning statements in the method are modified to show \texttt{ret$}$ being written to.

6.3.5 Statement

Statements within a method are represented using this node. If it is diagrammatically difficult to embed the actual program statement in the node, unique numbers may be assigned to program statements and those numbers embedded in this node. This trick helps avoid symbol clutter.
6.3.6 Data member

Data members of a class or local variables of a method can be conveniently represented by this node. This is linked to the corresponding owner by the membership link arc.

6.3.7 Annotation

This node may be viewed as an adornment on the basic CAG. Language-specific features and details can be captured using this node. The following are some possibilities:

- Type of a data member
- Accessibility of an element, i.e., *private*, *public*, etc.
- Derivation type in C++ (public, private, protected, virtual)
- Interfaces in Java
- Qualifiers *volatile*, *static* applied to elements of a class

6.3.8 Annotation link

An annotation node is connected to the corresponding annotated element by this link.

6.3.9 Element access

When one element needs access to another element, it is depicted using an element access link. It is a directed arc conveying the direction of access. For example, if a statement writes to a data member, the arrow points from the statement node to the data member node. For read access, the direction is reversed.
6.3.10 Control flow

Control flow arcs connect statements within a method. This represents an ordered sequence. Two arcs emanate with qualifiers $T$ and $F$ emanate from a conditional statement, to depict the branch taken when the condition is true or false respectively.

6.3.11 Membership link

This undirected arc connects a node with another node that it owns. For example, the class header will be connected to its methods and instance (or class) variables by this link. It is also used to model local variables of a method.

6.3.12 Inheritance

This directed arc points from a derived class to its parent. By default, public inheritance is assumed. However, it may be suitably annotated to capture other details.

6.3.13 Method call

The call site corresponding to a method is connected to its actual definition by this directed arc. The call site node looks the same as method definition, except for the arguments and return value; the actual arguments passed and the variable that gets the returned value are shown in this case.

Figure 6.2 depicts the CAG for the following Java class:

```java
class Stack {

    // Exported methods
    public Stack(int max_sz) {
        s_top = -1; elems = new Object[max_sz];
    }
    public void push(Object obj) {
        if( s_top < elems.length )
            elems[++s_top] = obj;
    }
}
```
public Object pop() {
    if (s_top > -1)
        return elems[s_top--];
    return null;
}

// Private implementation details
private int s_top;
private Object[] elems;
To understand how CAG can be used to represent inter-class method calls, consider the StackDriver class:

```java
class StackDriver {
    public static void main(String[] args) {
        Stack s1 = new Stack(10);
        X x1; // Assume existence of class X
        s1.push(x1);
        X x2;
        x2 = s1.pop()
    }
}
```
Figure 6.3 shows the corresponding CAG. As an example of representing language-specific features, consider Java's `interface`. Implementing an interface is different from deriving from a base class. In CAG, this is shown as an annotation, instead of using a language-specific visual element. For example, consider the following Java code:

```java
interface Cloneable {
    // Empty
}
class Employee implements Cloneable {
    // Details omitted
}
```

Figure 6.4 shows how this is modeled in CAG. Another example is the notion of private derivation supported in C++. An incomplete code fragment to implement a Stack using a List is given below, with the corresponding CAG in Figure 6.5.

```cpp
class List {
    public:
        void insert(const Object &obj);
        void append(const Object &obj);
        void delete(const Object &obj);
        const Object &deleteFirst();
        bool search(const Object &obj);
        int size() const;
    private:
        // implementation
    };
class Stack : private List {
    public:
        void push(const Object &obj) {
            List::insert(obj);
        }
        const Object &pop() {
            return List::deleteFirst();
        }
    }
```
Figure 6.3 CAG for Stack and StackDriver
Figure 6.4 Implementing an Interface (Java)

```cpp
bool isEmpty() {
    return List::size() == 0;
}
bool isFull() {
    return false;
}
```
Figure 6.5 Stack Derived from List
6.4 CONVERTING TO OTHER REPRESENTATIONS

It may sometimes be useful to convert a class annotated graph to another representation in order to apply an algorithm designed for use with that representation. Since CAG may be viewed as an extension of existing schemes, this is rarely a challenge. The following are three essential steps involved in the conversion:

1) Omit annotation nodes
2) Select appropriate subset of CAG elements
3) Map each applicable element of CAG to the target representation element

As an example, Figure 6.6 shows the Class Hierarchy Graph (Harrold 1996) corresponding to the annotated graph of Figure 6.5.

![Class Hierarchy Graph](image-url)
6.5  CAG AND LEXICAL FUNCTION CONTEXT

The LFC of a class member function (described in Chapter 2) is the set of all symbols this function binds to statically. For example, consider the C++ code fragment shown below:

```cpp
void g(int v) {
    // Details omitted
}
class X {
private:
    int a;
    int f();
public:
    X(int val) : a(val) {
    }
    void h() {
        f();
    }
};
int X::f() {
    return a;
}
```

The LFCs of member functions of X are

\[
\text{LFC}(X::\text{void}) = \{ X::\text{int}_a \}
\]

\[
\text{LFC}(X::\text{int}\_f\_\text{void}) = \{ X::\text{int}_a \}
\]

\[
\text{LFC}(X::\text{void}\_h) = \{ X::\text{int}_f\_\text{void}, ::\text{void}\_g\_\text{int} \}
\]

The LFC provides hint as which elements of to an evolved class may need to be retested. If the LFC of a member function after modification is different than the certified one, clearly retesting is called for. LFC may easily be derived from class annotated graphs by enumerating all the symbols referenced from within a method. CAG depicts symbol references in two ways:

- symbol accessed for read (use)
- symbol accessed for write (def)

196
For LFC computation, both *def* and *use* of a symbol in a method are taken to mean *referencing*. Figure 6.7 is the CAG for the above C++ code fragment. It is easy to see that the LFCs match *def-use* binding captured in CAG.

![Figure 6.7 CFG and Lexical Function Context](image)

### 6.6 SUMMARY

*Class Annotated Graphs* are an alternative to existing graphical representations of object-oriented software. Their primary advantage is the ability to capture, via *annotations*, language-specific details in addition to the normally required information. Such information can be used by appropriate algorithms to reason about the need to retest an evolved class. For instance, *lexical function context* of a method can be readily computed from a CFG. It is also possible to translate CFG to an existing representation formalism by removing annotations and mapping symbols to the target scheme. An example of such a conversion to *class hierarchy graphs* has been presented.