

A Dependence Representation for Coverage Testing of Object-Oriented Programs

E.S.F. Najumudheen^a Rajib Mall^a Debasis Samanta^b

- a. Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur, India
- b. School of Information Technology
Indian Institute of Technology, Kharagpur, India

Abstract We propose a dependence-based representation for object-oriented programs, named *Call-based Object-Oriented System Dependence Graph* (COSDG). Apart from structural features, COSDG captures important object-oriented features such as class, inheritance, and polymorphism. Novel features of COSDG include details of method visibility in a derived class, and different types of method call edges to distinguish between various calling contexts—simple, inherited, and polymorphic. We also propose an algorithm for the construction of COSDG, and subsequently explain its working with an example. COSDG has been developed primarily to aid test coverage analysis. However, it can be used in a variety of other software engineering applications such as program slicing, software re-engineering, and debugging.

Keywords Coverage analysis, program representation, software testing, object-oriented programs.

1 Introduction

Various software techniques such as test coverage analysis, program slicing, program debugging, software re-engineering, and compiler optimization convert a program into a suitable intermediate representation by code analysis, and use it for subsequent operations. An intermediate representation is essentially a model of a program that captures those characteristics that are relevant to a specific task while abstracting out the rest.

A variety of models have been proposed in the past to represent various features of programs. *Control flow graph* (CFG) [2, 6], *data flow graph* (DFG) [16, 6], *program dependence graph* (PDG) [6], *system dependence graph* (SDG) [8], and *call graph* (CG)

[20] are some of the well known representations. Each of these captures some specific features of a program: CFG depicts the flow of control between various program elements, DFG depicts data flow information between various program elements, PDG captures both control and data dependences for a single procedure, SDG represents dependences and procedure calls between multiple procedures, and CG represents calling relationships between various modules of a program.

These models were proposed to represent procedural programs. However, they cannot be used satisfactorily for object-oriented programs since the object-oriented paradigm introduces several features such as encapsulation, inheritance, polymorphism and dynamic binding.

In the past, several researchers have proposed extensions to dependence-based representations such as PDG and SDG, to incorporate features specific to object-oriented programs. Some of them were intended to meet the specific needs of a particular application [17, 10, 11], whereas some others were intended to support a variety of applications [13, 7]. Rothermel and Harrold proposed the *class dependence graph* (CIDG) [17]. Larsen and Harrold proposed a *system dependence graph for object-oriented software* (ESDG) [10]. Liang and Harrold proposed extensions to ESDG for the purpose of object-slicing [11]. Malloy *et al.* proposed a layered representation, the *object-oriented program dependency graph* (OPDG) [13]. Harrold and Rothermel proposed a family of representations for object-oriented software: *class hierarchy graph* (CHG), *class call graph* (CCG), *class control flow graph* (CCFG), *class dependence graph* (CIDG), and *framed graph* [7].

Although SDG and its modified versions have been used as intermediate representations for various software engineering applications [17, 10, 11, 23, 22, 9], attempts to perform test coverage analysis of object-oriented programs based on a dependence-based representation are scarcely reported in the literature. Earlier work have used control flow graphs, data flow graphs, def-use graphs, and call graphs as intermediate representations to perform test coverage analysis [4, 21, 1, 12, 18, 19], but not a dependence-based graph. Moreover, to perform an object-oriented coverage analysis, we need a representation that has the following aspects: be capable of capturing important object-oriented features, should help track the coverage of various program elements and features during execution, and facilitate accurate and efficient computation of object-oriented coverage measures. SDG or its modified versions either lack some of these aspects or possess unnecessary details (discussed in Section 6), and hence, cannot be used satisfactorily for test coverage analysis. Therefore, a specific representation is needed.

In this paper, we propose a dependence-based representation, based on ESDG, for test coverage analysis of object-oriented programs. We have named our representation *Call-based Object-Oriented System Dependence Graph* (COSDG). Our representation incorporates *dependence*, *flow*, *call graph*, and *inheritance* details. Dependence details include control dependence, data dependence, and membership dependence. Flow details include control flow and data flow. Call graph details consists of simple, inherited, and polymorphic method calls. Inheritance details consists of inheritance hierarchy among classes, and method visibility in a derived class. Though COSDG was developed specifically for coverage analysis operations, it has all the essential features needed for use in a variety of other software engineering applications such as program slicing, software re-engineering, and program debugging.

The rest of the paper is organized as follows. Section 2 provides the basic details needed to understand our representation. Section 3 describes our proposed represen-

tation, the Call-based Object-Oriented System Dependence Graph (COSDG). Section 4 describes the construction of the graph with the help of an example. Section 5 briefly describes our coverage analysis technique using the COSDG, and also discusses the results of our experimental study. We compare our work with related work in Section 6. Section 7 concludes this paper.

2 Background

In this section, we first provide a few definitions that would help the reader to understand the subsequent discussions. Next, we give an overview of SDG, CIDG, and then ESDG, which forms the basis of our proposed representation, COSDG.

Definition 2.1. Control Dependence: For two statements X and Y in a program, if Y is *control dependent* on X , then X must have two exit paths; one of the exit paths always results in Y being executed, and the other exit path may result in Y not being executed [6].

Definition 2.2. Data Dependence: For two statements X and Y in a program, Y is *data dependent* on X , if X defines a variable v , Y uses v , and there exists a directed path from X to Y along which there is no intervening definition of v [8, 17].

2.1 System Dependence Graph

The *System Dependence Graph* (SDG) is an extension of the *program dependence graph* [6], and represents a program that consists of multiple procedures and involves procedure calls. An SDG includes a *program dependence graph* to represent a system's main program, *procedure dependence graphs* to represent a system's auxiliary procedures, and some additional edges to interconnect these graphs [8].

In an SDG, a method call statement in a program (the corresponding program point is referred to as a *call site*) is represented by using a *call-site* vertex. Parameter passing between a call site and a called procedure is modeled by the introduction of four types of *parameter* vertices: *formal-in*, *formal-out*, *actual-in*, and *actual-out* vertices. A formal-in vertex is used to represent each formal parameter of the procedure, and a formal-out vertex is used to represent each formal parameter that may be modified by the procedure. Similarly, an actual-in vertex is used to represent each actual parameter at the call site, and an actual-out vertex is used to represent each actual parameter that may be modified by the called procedure. Formal-in and formal-out vertices are control dependent on the entry vertex of the procedure, whereas actual-in and actual-out vertices are control dependent on the call-site vertex.

A *call* edge is used to connect a call vertex to entry vertex of the called procedure. A *parameter-in* edge is used to connect an actual-in vertex to a formal-in vertex, and represents data flow from a call temporary¹ to a formal parameter. A *parameter-out* edge is used to connect a formal-out vertex to an actual-out vertex, and represents data flow from a formal parameter to a return temporary. In a procedure call, the value of an actual parameter represented by an actual-out vertex may depend on the value of another actual parameter represented by an actual-in vertex. Such a

¹An intermediate temporary variable created for each parameter, to effect the transfer of value between a call site and a called procedure [8].

dependence, termed as *transitive dependence*, is represented by using a *transitive dependence* edge to connect the actual-in to the actual-out vertex.

Data dependence edges are used to represent data flow between two statements within a method. Let two statements in a method be represented by vertices v_1 and v_2 . If vertex v_2 is *data dependent* on vertex v_1 , then v_1 is connected to v_2 by a data dependence edge.

2.2 Class Dependence Graph

The *Class Dependence Graph* (CIDG) represents the control and data dependencies within a class [17]. For a given class, the CIDG consists of a set of *program dependence graphs* (PDGs) [6] with additional edges to represent inter-procedural control and data dependencies. Each method (procedure) in a class is represented by an individual PDG. Hence, in a CIDG, each PDG is actually a *procedure dependence graph*.

Each procedure dependence graph contains an *entry vertex* that represents entry into a procedure. A statement in a procedure is represented by a *statement vertex*. Control and data dependencies between program statements are represented by *control dependence* and *data dependence* edges, respectively. For example, a control dependence edge from a vertex A to a vertex B implies that the statement represented by B is control dependent on the statement represented by vertex A (similarly for data dependence).

A *representative driver node* (RDN) serves as the *root* of the graph, and summarizes the set of drivers for class testing. Each `public` method in a class (represented as a PDG) is made a child of the *root*, by adding a *driver edge* from the *root* to the entry vertex of the PDG of that method.

A *state* vertex summarizes variables that make up the state of an object of a class. A state vertex is also made a child of the *root* vertex. The location of a method call in the program is referred to as a *call site*. A call to a method is represented by a *call edge* which connects a call site to the entry vertex of the called method.

2.3 Extended System Dependence Graph

Larsen and Harrold extended the *System Dependence Graph* to represent object-oriented programs [10]. In this paper, we refer to this graph as *Extended System Dependence Graph* (ESDG). Since an object-oriented software consists of a group of interacting classes, ESDG uses a class dependence graph (CIDG) to represent each class in a system. In an ESDG, the *root* node in the original CIDG is replaced by a *class entry vertex* which uniquely identifies a class, and *driver edges* are replaced by *class member edges*. A method in a class is represented by a *method dependence graph* which is similar to the procedure dependence graph discussed in CIDG. Class member edges connect a class entry vertex to the *method entry vertex* of each method in a class.

A *call site* in a method is represented as a *call vertex*. Parameter passing is modeled similar to the SDG with the introduction of parameter vertices and parameter edges. The *transitive dependence* edge in SDG is called a *summary* edge in ESDG. Moreover, since instance variables of a class are accessible to all methods in a class, a formal-in and a formal-out vertex is created for each instance variable that is referenced in a method.

For a derived class, the representation of the base class method is reused for an inherited method. Apart from connecting the class entry vertex of a class to the

method entry vertices of locally defined methods, class member edges also connect it to the method entry vertices of the methods inherited by the derived class.

A method call is termed as a *polymorphic method call* if there are several possible destinations of the call, and the actual destination is determined dynamically. ESDG uses a *polymorphic choice vertex* to represent the dynamic choice among the possible destinations of a polymorphic call. A *polymorphic* call vertex is connected to a polymorphic choice vertex by a call edge. Calls to each possible destination is represented by a subgraph, and call edges are used to connect the polymorphic choice vertex to the individual subgraphs.

3 Call-based Object-Oriented System Dependence Graph (COSDG)

In this section, we present our dependence-based representation for object-oriented programs, named Call-based Object-Oriented System Dependence Graph (COSDG). The COSDG is based on ESDG. Like ESDG, each class in a COSDG is represented by a class dependence graph, but those aspects that are not needed for test coverage analysis (e.g., polymorphic choice vertex) are excluded from the representation. Moreover, it incorporates *visibility* details of inherited methods in the representation, and represents *polymorphic* and *inherited* method calls differently. These modifications are intended to help achieve accurate polymorphic and inheritance coverage measures for object-oriented programs.

COSDG is a directed, connected multigraph $G = (V, E)$, consisting of a set V of vertices and a set E of edges. A vertex $v \in V$ represents one of the three categories of vertices, namely, *statement vertices*, *entry vertices*, and *parameter vertices*. An edge $e \in E$ represents one of the seven categories of edges, namely, *control dependence edges*, *data dependence edges*, *parameter dependence edges*, *method call edges*, *summary edges*, *class member edges*, and *inheritance edges*.

In the discussions to follow, we have provided a pictorial view of the graphs (partial or subgraphs of COSDG) representing the programs and code snippets given as examples. The graphical representation for different types of vertices and edges in a COSDG are shown in Figure 1. For clarity, vertices are labeled with their suffixes. For example, vertices $v_{e1.01}$, $v_{s1.05}$, and $v_{p1.01}$ are labeled as **e1.01**, **s1.05**, and **p1.01**, respectively. The different types of vertices and edges are explained in the following sections.




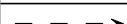

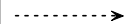

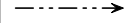

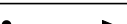
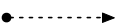

Representation for Vertices		Representation for Edges	
	class entry, method entry		control dependence
	parameter		data dependence
	statement, call		parameter dependence
			class member
			summary
			inheritance
			simple method call
			inherited method call
			polymorphic method call

Figure 1 – Graphical Representation for different types of vertices and edges in a COSDG

3.1 Vertices

A vertex is denoted as $v_{t.n}$ ($v_{t.n} \in V$) where t specifies the type of a vertex and n is an integer suffix that uniquely identifies a vertex. Program statements within

the body of a method are represented by *statement* vertices (V_s). These are of two types, namely, *simple statement* vertices and *call* vertices. Statements that invoke a method (*call sites*) are represented by call vertices (V_{s2}), whereas program statements other than method calls, such as assignments, loops, and conditions, are represented by simple statement vertices (V_{s1}). Class and method headers are represented by *entry* vertices (V_e): class headers by *class entry* vertices (V_{e1}), and method headers by *method entry* vertices (V_{e2}). Hence, $V_s = V_{s1} \cup V_{s2}$, and $V_e = V_{e1} \cup V_{e2}$.

COSDG adopts the ESDG's model for parameter passing between a caller and a callee². It is modeled by using *parameter* vertices (V_p) and *parameter dependence* edges (Section 2.1). There are four types of parameter vertices, namely, *formal-in* (V_{p1}), *formal-out* (V_{p2}), *actual-in* (V_{p3}), and *actual-out* (V_{p4}) vertices. These vertices are similar to the *parameter* vertices mentioned in Section 2.1.

In the discussions to follow, $v_{s1.n} \in V_{s1}$, $v_{s2.n} \in V_{s2}$, $v_{e1.n} \in V_{e1}$, and $v_{e2.n} \in V_{e2}$ denote a simple statement, a call, a class entry, and a method entry vertex, respectively. Similarly, $v_{p1.n} \in V_{p1}$, $v_{p2.n} \in V_{p2}$, $v_{p3.n} \in V_{p3}$, and $v_{p4.n} \in V_{p4}$ denote a formal-in, a formal-out, an actual-in, and an actual-out vertex, respectively.

3.2 Edges

An edge is denoted as $e_{t.n}$ ($e_{t.n} \in E$) where t specifies the type of an edge, and n is an integer suffix that uniquely identifies an edge. Passing of values between actual and formal parameters is represented by *parameter dependence* edges (E_p), which are of two types: *parameter-in* (E_{p1}) and *parameter-out* (E_{p2}) edges. *Data dependence edges* (E_d) represent the flow of data between different statement vertices of the COSDG. These edges are similar to the *parameter* and *data dependence* edges described in Section 2.1. *Summary edges* (E_s) represent the transitive flow of dependence between actual-in and actual-out vertices.

Thus, $e_{p1.n} \in E_{p1}$, $e_{p2.n} \in E_{p2}$, $e_{d.n} \in E_d$, and $e_{s.n} \in E_s$ denote a *parameter-in*, a *parameter-out*, a *data dependence*, and a *summary* edge, respectively. Other types of edges are explained in the following subsections.

3.2.1 Class Member Edges

Class member edges (E_b) are used to represent the membership relation between a class and its methods. They associate all locally defined and overriding methods of a class with the class entry vertex. A class entry vertex is connected to a method entry vertex by using a class member edge. It is denoted as $e_{b.n}$ where b specifies a class member edge, and n is an integer suffix that uniquely identifies the edge.

The Java program shown in Figure 2(a) has three classes A, B, and C consisting of three, four, and two methods, respectively. Figure 2(b) shows the connection between each class and its member methods by class member edges.

3.2.2 Inheritance Edges

Inheritance edges (E_i) represent the inheritance relation between classes. An inheritance edge connects a child class to its parent class in the direction of the inheritance dependence. It is tagged with a list of methods of a parent class that are visible in its child class, i.e., method declared as **protected** and **public** in a parent class, but not

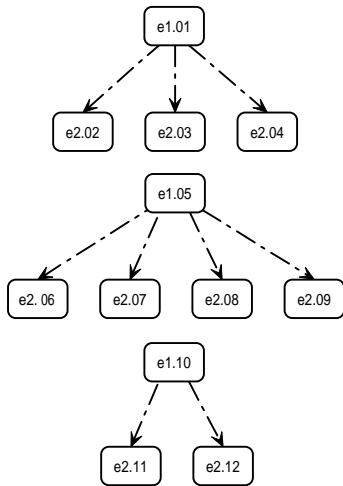
²We use the terms 'caller' and 'callee' to refer to a 'calling statement' (a call site) and a 'called method', respectively.

```

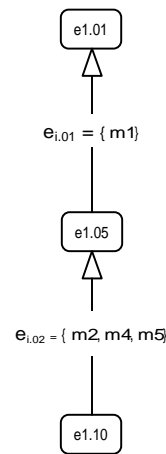
01: class A {
02:   private void m0(){
      // base
03:   protected void m1(){
      // base
04:   public void m2(){
      // base
05:   }
06: class B extends A {
07:   public void m2(){
      // overriding
08:   private void m3(){
      // locally defined
09:   }
10: class C extends B {
11:   private void m6(){
      // locally defined
12:   public void m7(){
      // locally defined
13:   }

```

(a) An example Java program



(b) Class member edges



(c) Inheritance edges with tags showing method visibility

Figure 2 – (a) A Java program depicting class membership and inheritance dependences (for clarity, only relevant edges are shown)

overridden in a child class. An inheritance edge is denoted as $e_{i,n}$ where i specifies an inheritance edge, and n is an integer suffix that uniquely identifies the edge.

Figure 2(c) illustrates the inheritance hierarchy of classes A, B, and C for the Java program in Figure 2(a). In the example, class B is derived from class A, and class C is derived from class B. Hence, class entry vertex $v_{e1.05}$ is connected to class entry vertex $v_{e1.01}$ by an inheritance edge $e_{i,01}$, and vertex $v_{e1.10}$ is connected to vertex $v_{e1.05}$ by another inheritance edge $e_{i,02}$. Out of the three methods defined in class A, m_0 is declared as *private*, m_1 as *protected*, and m_2 as *public*. However, as method m_2 is overridden in the derived class B, only method m_1 is visible in class B. Hence, edge $e_{i,01}$ is tagged with m_1 only. Similarly, edge $e_{i,02}$ is tagged with m_2 , m_4 , and m_5 .

3.2.3 Control Dependence Edges

Control dependence edges (E_c) represent control conditions on which the execution of a program element depends. A control dependence edge is used to connect a pair of vertices, say v_1 to v_2 , if v_2 is *control dependent* on v_1 . Pairs of program elements in which the second element is control dependent on the first element is listed below.

- A.1. A method and a statement defined within its body.
- A.2. An iterative (loop) or a conditional statement and a statement nested within the loop or condition.
- A.3. A statement and itself (indicates a loop).
- A.4. A method and its formal parameter.
- A.5. A call site and its actual parameter.

An control dependence edge is denoted as $e_{c.n}$ where c specifies a control dependence edge, and n is an integer suffix that uniquely identifies the edge.

Figure 3 depicts the various instances of control dependence in a program. From the Java code in Figure 3(a), it can be noted that the statements in line 11, 12, 14, and 16 are directly dependent on the statement in line 10 (method header). Hence, the method entry vertex $v_{e1.10}$ is connected to statement vertices $v_{s1.11}$, $v_{s1.12}$, $v_{s1.14}$, and $v_{s1.16}$ by control dependence edges in Figure 3(b) (Clause A.1). The statements in line 13 and line 15 are dependent on the `while` statement in line 12 and `if` statement in line 14 respectively. Therefore, statement vertices $v_{s1.12}$ is connected to $v_{s1.13}$, and $v_{s1.14}$ is connected to $v_{s1.15}$, by control dependence edges (Clause A.2). The edge from $v_{s1.12}$ to itself indicates a loop (Clause A.3). Vertices $v_{p1.01}$ and $v_{p1.02}$ represent the formal parameters `x` and `y` of method `mult`, and vertex $v_{p2.03}$ represent the value returned by `mult`. Hence, entry vertex $v_{e1.10}$ is connected to vertices $v_{p1.01}$, $v_{p1.02}$, and $v_{p2.03}$ by control dependence edges (Clause A.4). Similarly, call vertex $v_{s2.60}$ representing the call to `mult` in line 60, is connected to actual parameter vertices $v_{p3.04}$, $v_{p3.05}$, and $v_{p4.06}$ (Clause A.5).

3.2.4 Method Call Edges

Method call edges (E_m) are of three types, namely, simple call edges (E_{m1}), inherited call edges (E_{m2}), and polymorphic call edges (E_{m3}). We introduce a few definitions prior to describing different types of method call edges.

Definition 3.1. *Inherited method:* A method in a derived class that is inherited from one of its ancestor classes, but not overridden in that class is called an inherited method.

Definition 3.2. *Non-inherited method:* A method that is locally defined in a class or that overrides a method in an ancestor class is called a non-inherited method.

Definition 3.3. *Sender and receiver objects:* A *sender object* is an object that sends a request (a message) to another object. A *receiver object* is an object that receives a message from another object. Both sender and receiver may be the same object.

Definition 3.4. *Receiver class:* The class of a receiver object is called a receiver class.

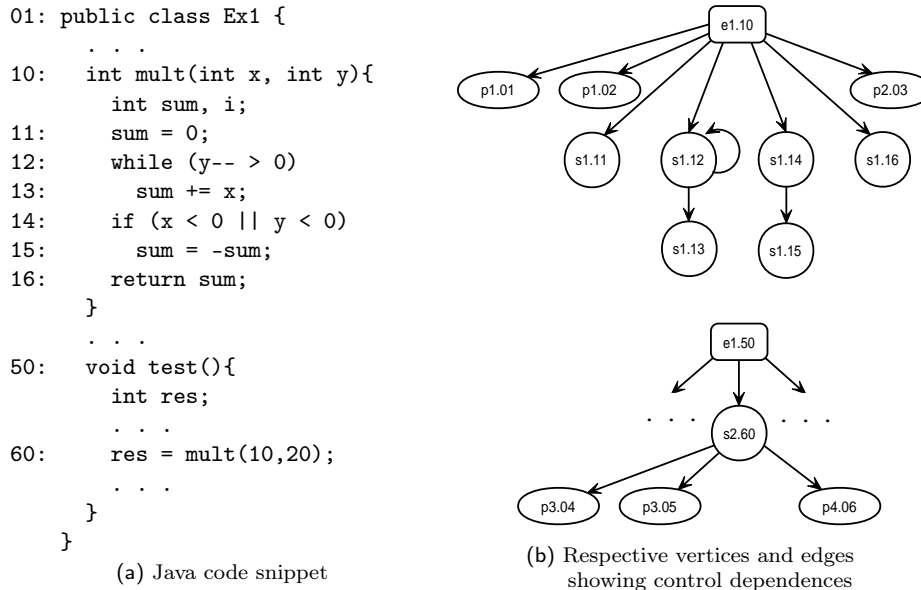


Figure 3 – Various instances of control dependence in a COSDG (only relevant edges are shown)

Definition 3.5. Candidate receiver class: A receiver class that is *feasible* to be bound to a receiver object at a polymorphic call site is called a candidate receiver class.

Definition 3.6. Target method: The method invoked by a receiver object as a reaction to a message is called a target method.

Definition 3.7. Candidate target method: A target method that is *feasible* to be invoked by a receiver object at a polymorphic call site is called a candidate target method.

Simple Call Edge

A simple call edge connects a call site to a method defined in the class of an invoking object. It is denoted as $e_{m1.n}$ where $m1$ specifies a simple method call, and n is an integer number that uniquely identifies the call edge. An edge is designated as a simple call edge in each of the following situations.

- B.1. A constructor of a class is called.
- B.2. An object reference calls its non-inherited method.
- B.3. A method calls itself (recursion) or another method of its own class.
- B.4. A reference to an object calls an inherited method (m_1), which in turn calls another overriding method (m_2) in the receiver class at a call site C_k . Then, the call at C_k adds a new simple call edge from C_k to m_2^3 , in addition to other edges that may exist at C_k .

³Since m_2 is overridden, m_2 defined in the receiver object is called. Hence, the call would be a simple call.

Class **X** in the example Java program shown in Figure 4 depicts the different instances of a simple method call. Figure 5(a) illustrates the creation of simple call edges at the respective call sites in method **mx**.

Call sites **c1** and **c2** in method **mx** are calls to constructors **B()** and **C()** (Clause B.1). Therefore, we have simple call edges, $e_{m1.01}$ and $e_{m1.02}$ from vertices $v_{s2.19}$ and $v_{s2.20}$ to entry vertices $v_{e2.04}$ and $v_{e2.13}$ respectively. The call to method **m1** by **Obj_B** is a simple call (call site **c3**) since **m1** is a non-inherited method in **B** (Clause B.2). Hence, call vertex $v_{s2.21}$ is connected to method entry vertex $v_{e2.05}$ by a simple call edge $e_{m1.03}$. Method **m1** in turn calls method **m2** (call site **c4**). Since **m2** is defined locally in class **B**, it is a simple call (Clause B.3). Therefore, call vertex $v_{s2.06}$ is connected to entry vertex $v_{e2.07}$ by a simple call edge $e_{m1.04}$. Finally, **Obj_C** calls **m1**, a method inherited from its parent **B**. This call is an inherited method call (discussed in the following section). However, as method **m2** is overridden in class **C** (receiver class), in the current context, the overriding method **m2** is called at call site **c4** (Clause B.4). This call is represented as a simple call edge $e_{m1.06}$ from call vertex $v_{s2.06}$ to method entry vertex $v_{e2.14}$.

Inherited Call Edge

An inherited call edge connects a call site to a method inherited by the class of an invoking object. It is denoted as $e_{m2.n}$ where $m2$ specifies an inherited method call, and n is an integer number that uniquely identifies the call edge. An edge is designated an inherited call edge in each of the following situations.

- C.1. An object reference calls its inherited method.
- C.2. A method calls a method of its super class.
- C.3. A reference to an object calls an inherited method (m_1), which in turn calls another inherited method (m_2) in the receiver class at call site C_k . Then, the call at C_k adds a new inherited call edge from C_k to m_2 , in addition to other edges that may exist from at C_k .

Class **Y** in the program shown in Figure 4 depicts the different instances of an inherited method call. Figure 5(b) illustrates the creation of inherited call edges at the respective call sites in method **my**.

After the calls to constructors **B()** and **C()** (call sites **c6** and **c7**), reference variable **Inh_B** calls method **m3** at call site **c8**. Since **m3** is a locally defined method, it is a simple method call. Method **m3** in class **B** calls method **m0** in class **A** (parent class) at call site **c9** (Clause C.2). Therefore, call vertex $v_{s2.09}$ is connected to method entry vertex $v_{e2.02}$ by an inherited method call edge $e_{m2.10}$. Next, reference variable **Inh_C** calls an inherited method **m4** at call site **c10** (Clause C.1). So, call vertex $v_{s2.28}$ is connected to entry vertex $v_{e2.10}$ by an inherited call edge $e_{m2.11}$. Method **m4**, in turn, calls method **m1**. As the receiver class is **C**, in the current context, the call to method **m1** is an inherited method call (Clause C.3). Hence, call vertex $v_{s2.11}$ is connected to entry vertex $v_{e2.05}$ by an inheritance call edge $e_{m2.12}$.

Polymorphic Call Edge

A polymorphic call edge connects a call site to a method defined in one of the candidate receiver classes. It is denoted as $e_{m3.n}$ where $m3$ specifies a polymorphic method call, and n is an integer number that uniquely identifies the call edge. An edge is designated as a polymorphic call edge in the following situation.

D.1. A reference to an object calls a method at a call site C_k . If the target method can not be determined statically, polymorphic call edges are added at C_k from the caller to the callee, for each candidate target method.

Class Z in the program shown in Figure 4 depicts a polymorphic method call. Figure 5(c) illustrates the creation of polymorphic call edges at the call site. Since the reference variable `Poly_B` at call site `c15` in method `mz` of Class Z can refer to an instance of one of the three classes, B, C, or D, the call to method `m2` by `Poly_B` is a polymorphic method call. Therefore, polymorphic call edges $e_{m3.16}$, $e_{m3.17}$, and $e_{m3.18}$ are added from the call vertex $v_{s2.37}$ to the method entry vertices $v_{e2.07}$, $v_{e2.14}$, and $v_{e2.14}$, respectively.

```

01: class A {
02:   void m0() {
03:     //base class method
04:   }
05: }
06: class B extends A {
07:   B() { //constructor }
08:   void m1() {
09:     //locally defined - 1
10:     m2(); // (c4)
11:   }
12:   void m2() {
13:     //locally defined - 2
14:   }
15:   void m3() {
16:     m0(); // (c9)
17:   }
18:   void m4() {
19:     m1(); // (c11)
20:   }
21: }
22: class C extends B {
23:   C() { //constructor }
24:   void m2() {
25:     //overriding
26:   }
27: }
28: class D extends C {
29:   D() { //constructor }
30:   //inherited
31: }
32: class X {
33:   void mx() {
34:     B Obj_B = new B(); // simple (c1)
35:     C Obj_C = new C(); // simple (c2)
36:     Obj_B.m1(); // simple (c3)
37:     Obj_C.m1(); // inherited (c5)
38:   }
39: }
40: class Y {
41:   void my() {
42:     B Inh_B = new B(); // (c6)
43:     C Inh_C = new C(); // (c7)
44:     Inh_B.m3(); // simple (c8)
45:     Inh_C.m4(); // inherited (c10)
46:   }
47: }
48: class Z {
49:   void mz() {
50:     int x;
51:     . . .
52:     B Poly_B = new B(); // (c12)
53:     if (x == 10)
54:       Poly_B = new C(); // (c13)
55:     else
56:       if (x == 20)
57:         Poly_B = new D(); // (c14)
58:     Poly_B.m2(); //polymorphic (c15)
59:     . . .
60:   }
61: }

```

Figure 4 – An example Java program depicting different types of method calls

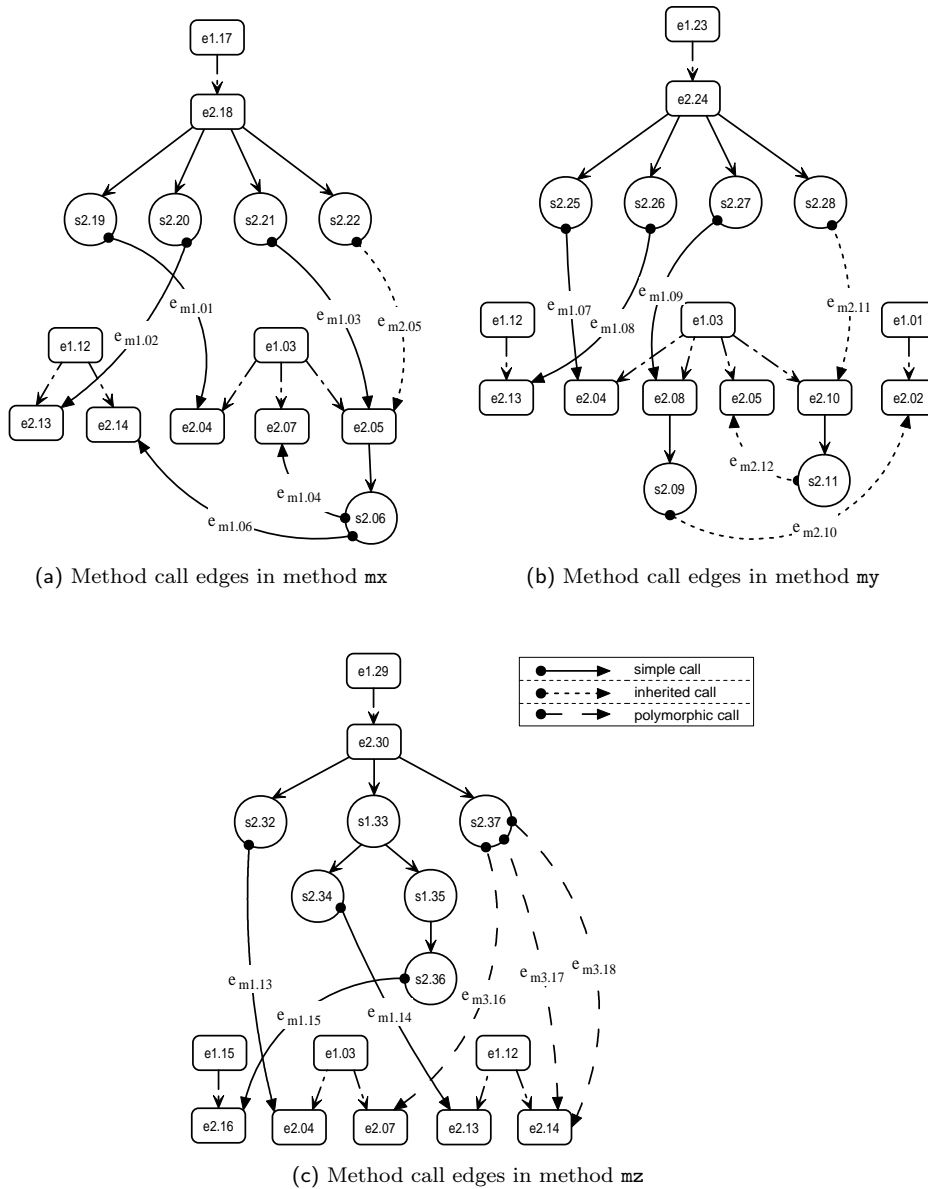


Figure 5 – Edges illustrating various types of method calls in the program shown in Figure 4 (only relevant vertices and edges are shown)

4 Construction of a COSDG

In this section, we discuss the construction of the COSDG representation of a program. We first outline the various steps in constructing the CODSG for a complete program. First, the class dependence graph for each class is constructed. Next, the inheritance hierarchy is established among classes by connecting the parent and child classes with inheritance edges. Finally, algorithm `BuildCallSite` processes the call sites in each

method to establish a connection between a call site and a callee, which results in a connected multigraph. This is done in two stages. First, call-sites in all the non-`main`⁴ methods are processed to build clusters. Next, call sites in the `main` method are processed to establish a connection between the class that contains the `main` method and the classes within each cluster. Thus, it builds a call graph for the complete program by incrementally adding method call edges at the call sites. These steps have been presented in pseudo-code form in Figure 6(a) (Algorithm `ConstructCOSDG`).

The pseudo code for algorithm `ConstructMDG` is shown in Figure 6(b). It outlines the different steps in constructing the method dependence graph for a method definition. First, the method header is processed and a method entry vertex is created. Subsequently, formal-in and formal-out parameter vertices are created. The parameter vertices are connected to the method entry vertex with control dependence edges. Next, the statements within the method definition are processed, and corresponding statement and call-site vertices are created. Then, after performing control dependence and data dependence analyses, control dependence and data dependence edges are added.

Algorithm `BuildCallSite` shown in Figure 6(c) outlines the different steps in processing a method call statement (call site). First, actual-in and actual-out parameter vertices are created at each call site. The parameter vertices are connected to the corresponding call vertex with control dependence edges. Next, data flow between a call site and its callee is established by adding parameter edges between actual and formal vertices. Then, summary edges are added to indicate transitive dependencies between actual-in and actual-out parameter vertices. Finally, various method call edges are added (described in Section 3.2.4).

Example

We now illustrate the construction of the COSDG with the help of an example Java program shown in Figure 7⁵. Class `A` is the base class. Class `B` is derived from class `A`, `C` is derived from `B`, and `D` is derived from `C`. The `main` method in class `Test` contains the test driver code.

The class dependence graph for each class is created in three steps. First, the class entry vertex for a class is created (Step 1a). In Step 1b(i), `ConstructCOSDG` calls `ConstructMDG` to construct the method dependence graphs for the methods of each class in the program. Then, the method dependence graphs are associated with the class entry vertex by adding class member edges (Step 1b(ii)). Figure 8(a) illustrates the creation of class entry vertices and the construction of method dependence graphs. In Step 2, `ConstructCOSDG` establishes the inheritance hierarchy. Figure 8(b) illustrates the addition of class member edges, and the construction of the inheritance tree. In Step 3, the algorithm invokes `BuildCallSite` to process the call sites. For simple method calls, a simple call edge is added from the call vertex to the method entry vertex of the callee. For polymorphic calls, a polymorphic call edge is added from the call vertex to the method entry vertices of each target method. Method calls from various call sites to their respective method entry vertices are shown in Figure 8(c). It shows the final form of the COSDG for the Java program given in Figure 7. To avoid cluttering, only some of the data dependence edges have been shown in the figure. Also, parameter and summary edges have not been shown in the figure.

⁴Here, we assume that the `main` method represents the *test driver* code that would test the various features of classes.

⁵A detailed description of the working of the algorithm `ConstructCOSDG` is available in [14].

Algorithm ConstructCOSDG**Input:** An Object-Oriented Program/* A set of n interacting classes */**Output:** COSDG

/* A dependence-based graph */

1. Construct *Class Dependence Graphs*
 - For each class
 - a. Create a class entry vertex
 - b. Construct *Method Dependence Graphs*
 - For each method
 - i. Construct a *method dependence graph* by calling **ConstructMDG**
 - ii. Add a class member edge /* From class entry to method entry */
2. Build inheritance hierarchy
 - Add *inheritance edges* to COSDG
3. Process call sites
 - a. Perform **BuildCallSite** for non-main methods
 - b. Perform **BuildCallSite** for the main method
- (a) Algorithm to construct the COSDG for an object-oriented program

Algorithm ConstructMDG**Input:** A Method**Output:** A Method Dependence Graph

1. Process method header
 - a. Create a method entry vertex
 - b. Create formal parameter vertices /*Add formal-in, formal-out vetices*/
 - c. Add control dependence edges /* Method entry vertex to parameter vertices */
 2. Create statement vertices and call-site vertices
 3. Determine control dependences between vertices; Add control dependence edges
 4. Determine data dependences between vertices; Add data dependence edges
- (b) Algorithm to construct a Method Dependence Graph (subgraph) for a method

Algorithm BuildCallSite**Input:** Partially constructed COSDG, a Method**Output:** COSDG with method calls added

1. At each call site
 - a. Create actual parameter vertices /*Add actual-in, actual-out vertices */
 - b. Add control dependence edges /* Call vertex to parameter vertices */
 - c. Add a parameter edge for each pair of actual/formal vertices
/*Add parameter-in, parameter-out edges */
 - d. Determine transitive dependences; Add summary edges
 - e. Process method calls; Add method call edges
- (c) Algorithm to process a method call at a call site

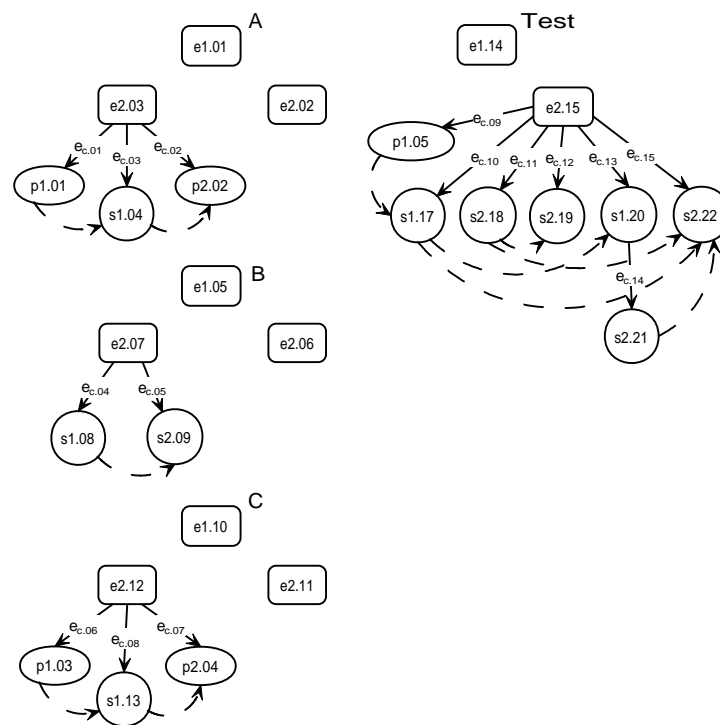
Figure 6 – (a) Algorithms to construct the COSDG and its subgraphs

```

01: class A {
02:   A() { //constructor }
03:   int m2(int x) {
04:     return x+1;
05:   }
06: }
07: class B extends A {
08:   B() { //constructor }
09:   void m1() {
10:     int y = 10;
11:     y = m2(y);
12:   }
13: }
14: class C extends B {
15:   C() { //constructor }
16:   int m2(int x) {
17:     return ++x;
18:   }
19: }
20: public class Test {
21:   public static void
22:     main(String[] args) {
23:     //test driver code
24:     int x,y;
25:     x = args.length;
26:     B ObjRef = new B();
27:     ObjRef.m1();
28:     if (x == 1)
29:       ObjRef = new C();
30:     y = ObjRef.m2(x);
31:   }
32: }

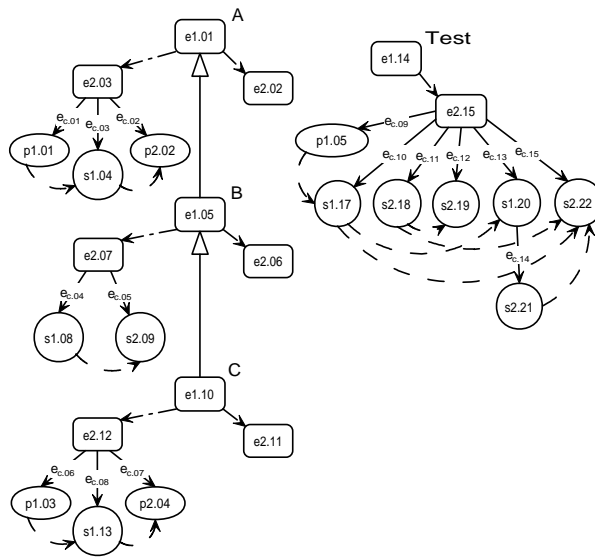
```

Figure 7 – An example Java program to illustrate the construction of the COSDG

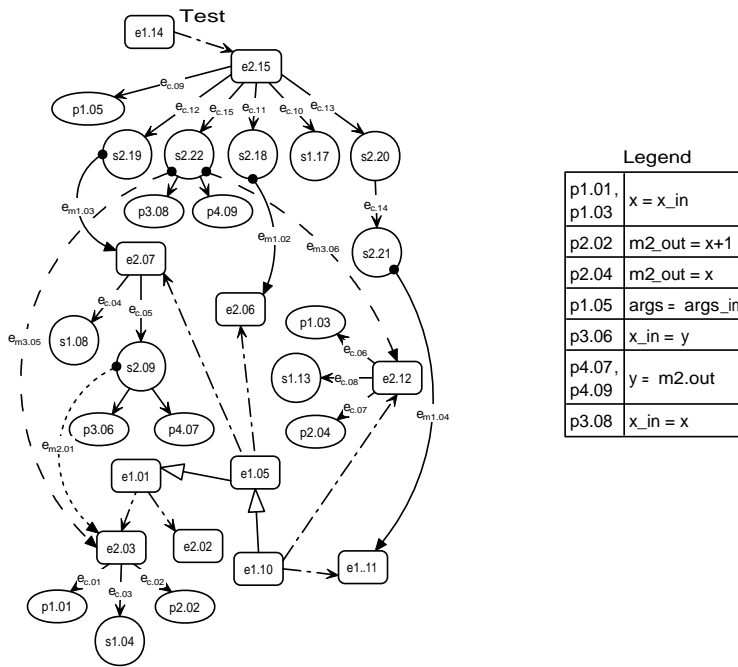


(a) Subgraphs created after Step 1b(i) of algorithm ConstructCOSDG

Figure 8 – Incremental construction of COSDG (Contd.)



(b) COSDG after completion of Step 2 of the algorithm



(c) COSDG after Step 3 of the algorithm

Figure 8 – Incremental construction of COSDG for the Java program in Figure 7.

5 Test Coverage Analysis using COSDG

In this section, we briefly describe our approach to test coverage analysis of object-oriented programs using our proposed representation, the COSDG [15]. Figure 9 gives the schematic of our test coverage analysis technique. First, the source code is parsed by a *graph builder* to construct the COSDG of the program under test. Next, the source code is instrumented by a *source code instrumenter* at particular program points depending on the criteria specified by the user. The instrumented code is then executed with different test inputs in an integrated run-time environment (for e.g., Java Runtime environment). The execution traces of various test runs form the input to a *graph marker*, which marks the edges of COSDG based on the executed features. Finally, the marking on the different types of edges in the COSDG are analyzed, and various coverage measures are computed by a *coverage analyzer*.

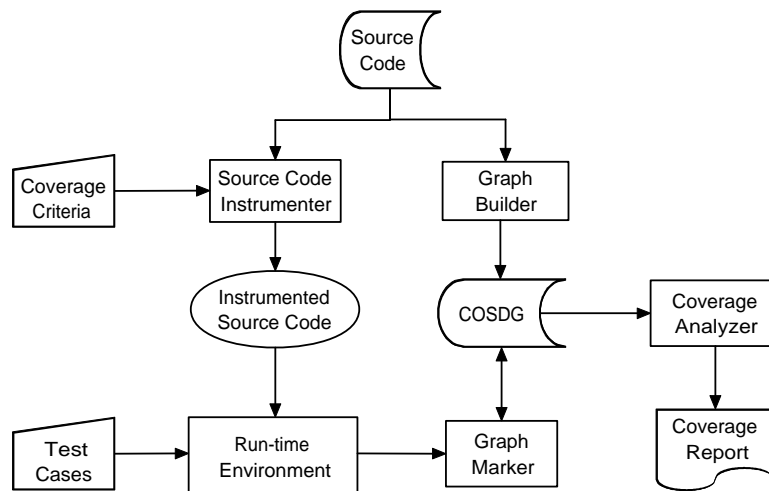


Figure 9 – Schematic of our test coverage analysis technique.

5.1 Experimental Results

Our test coverage analysis technique has been implemented in a prototype tool. It has been developed in Java using *Eclipse IDE* [5] and *ANTLR* tool [3]. An object-oriented program (presently, Java), coverage criteria, and a test suite form the input to the tool. It outputs a coverage report, which provides details of the various program features exercised by the test suite.

We have performed several experimental studies using the *Graph Builder* module for programs of varying sizes. The programs used for our study consisted of 5 to 20 classes, and the program size varied from 233 to 1712 statements. Table 1 summarizes the results of our experiments. The fourth and the fifth columns denote the number of vertices and edges created by algorithm `ConstructCOSDG`, respectively. The sixth column shows the memory required to store the COSDG constructed by the algorithm, and the last column provides the time taken to construct the graph. Typically, a base vertex requires 12 bytes, and a base edge requires 21 bytes of storage. The different types of vertices and edges of COSDG are derived from the base vertex and base edge respectively. The size of the vertices varies from 20 to 66 bytes, whereas the size of the edges varies from 37 to 113 bytes. The data structures for the base vertex

Table 1 – Graph size and time taken by algorithm `ConstructCOSDG`

Program	Classes (#)	Stmts. (#)	Vertices (#)	Edges (#)	COSDG Size (KB)	Construction Time (mSecs)
<i>P1</i>	5	233	291	365	30.0	51
<i>P2</i>	7	426	554	667	55.1	74
<i>P3</i>	8	613	842	989	81.4	97
<i>P4</i>	11	827	1103	1301	106.7	115
<i>P5</i>	13	998	1426	1613	134.3	128
<i>P6</i>	14	1224	1738	1924	161.9	152
<i>P7</i>	17	1453	2073	2240	189.6	173
<i>P8</i>	20	1712	2461	2657	224.8	201

```

public class Node {
    private int nodeId;
    private int nodeType;
    private int lineNo;
}

public class Edge {
    private int edgeId;
    private int edgeType;
    private Node srcNode;
    private Node desNode;
    private boolean mark = false;
}

```

Figure 10 – Data structures for a base vertex and a base edge in a COSDG

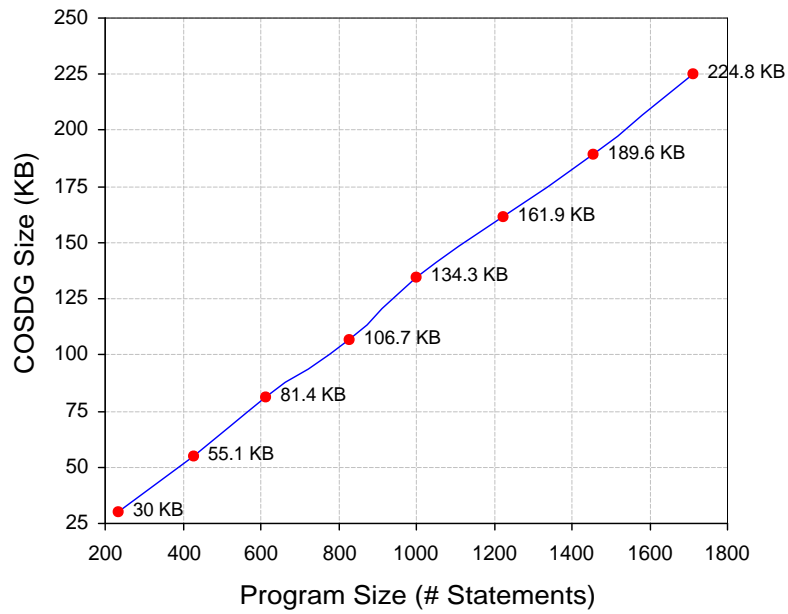
(Node) and the base edge (Edge) are shown in Figure 10. Apart from the memory required to store the COSDG, the tool also requires memory to store temporary data structures like class table, method table, binding sets, etc., which are used during graph construction. This amounts to approximately 3% of the size of the COSDG. However, this memory is reclaimed by the system at the end of graph construction, and hence, have not been included in the table.

From the results, we can observe that both the memory required to store the COSDG and the time taken to construct it, increase almost linearly with increasing program size. These results have been illustrated in Figure 11(a) and Figure 11(b). The linear increase in COSDG size is quite obvious as every program statement adds a vertex and one or more edges to the graph, depending on the vertex type. The linear increase in time can be attributed to the combined effect of two components: code parsing time and the time needed to create graph components.

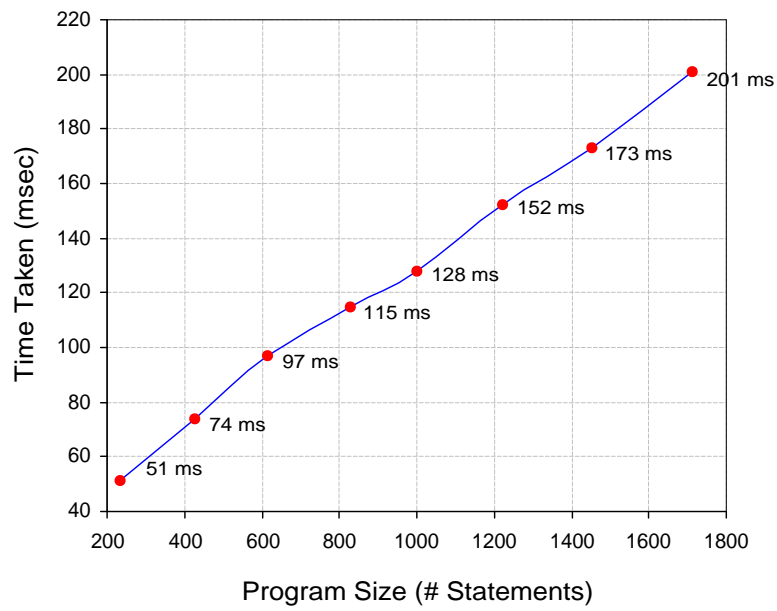
6 Related Work

In this section, we compare our representation with other similar representations for object-oriented software proposed in the past by various researchers.

Larsen and Harrold proposed the System Dependence Graph for object-oriented programs (ESDG) [10], which forms the basis of our representation. However, in our representation, we have incorporated several modifications to ESDG to make it suitable for test coverage analysis. First, COSDG represents a derived class differently. In ESDG, a class entry vertex of a derived class is also connected to the methods inherited by it from the base class, by class member edges. In COSDG, each class



(a) Increase in graph size with increasing program size



(b) Increase in graph construction time with increasing program size

Figure 11 – Memory required to store COSDG and time needed to construct COSDG by algorithm ConstructCOSDG for various programs

entry vertex is connected by class member edges to its locally defined methods only. Second, we have introduced the *inheritance edge* to represent the inheritance dependence between classes. Methods inherited by a derived class can be known by using the inheritance edge. Therefore, class member edges to inherited methods have been removed. Third, COSDG differs from ESDG in the way it models polymorphic method calls. COSDG does not have polymorphic choice vertices. Instead, at each polymorphic call site, it adds a polymorphic call edge from the call site to the entry vertex of each possible target method. These modifications were done to provide an efficient traversal of the COSDG and also achieve accurate coverage measures.

Rothermel *et al.* proposed the Class Dependence Graph (CIDG) [17] to select regression tests for modified or derived classes. CIDG uses a *driver node* as the root of the graph, and *driver edges* to connect the root to the methods (both local and inherited methods). Liang *et al.* modified ESDG to provide a representation for polymorphic objects used as parameters [11]. Their representation uses an object-flow subgraph to inspect statements in a slice, object by object (*object slicing*). Malloy *et al.* proposed the Object Program Dependency Graph (OPDG) based on the program dependence graph [13]. OPDG was designed to support applications such as profiling and debugging, and uses a generalized structure to represent the features of a program. Harrold *et al.* proposed a family of graph representations for object-oriented programs [7].

Since our primary aim is to build a suitable representation to aid test coverage analysis operations, we have designed the COSDG with the following features.

- Provide various object-oriented coverage measures like inheritance coverage and polymorphic coverage, in addition to the traditional measures.
- Provide accurate coverage measures by avoiding spurious dependencies (i.e., *infeasible* edges removed to the maximum extent).
- Support efficient traversal of the graph for marking and analysis.

To realize the above features, COSDG distinguishes various types of method calls, namely, simple, inherited, and polymorphic, with different types of method call edges, whereas none of the earlier work [17, 11, 13, 7] provide this feature. Furthermore, unlike in [7], COSDG does not add a *polymorphic call* vertex and a *return* edge to represent a polymorphic call and return. Return from a method call is implied in the COSDG. Moreover, COSDG neither use a *region vertex* nor a *control flow edge* to represent an explicit control flow in program constructs as in [13], as they are insignificant for the purpose of coverage analysis; our representation needs to capture the coverage of various elements of a program (loops, blocks, calls, etc.) by a test suite rather than the frequency of execution of elements.

Kovács *et al.* proposed a representation for Java programs based on ESDG [9]. In their representation, *class member* edges are tagged as *public*, *protected*, or *private* to indicate the visibility of a method. In contrast, COSDG tags the *inheritance dependence* edge with a list of methods that are visible to the subclasses. This is more suitable for coverage analysis, as we need to know whether each method visible to a subclass needs to re-tested or not.

Other Java-specific representations like Software Dependence Graph for Java (JSDG) proposed by Zhao [23] and Java System Dependence Graph (JSysDG) proposed by Walkinshaw [22] have also included Java-specific features like packages, interfaces, and abstract classes in the representations. Zhao's JSDG adopts the ESDG model to represent inheritance and polymorphism, and the Kovács model to represent *packages* and *interfaces*. On the other hand, JSysDG adopts Liang's modified

ESDG model to represent polymorphic objects, and Kovács model to represent the visibility of methods in a class. In addition, JSysDG differentiates between normal and abstract methods by using *abstract method* edges.

7 Conclusions

We have proposed a dependence-based representation for object-oriented programs, named *Call-based Object-Oriented System Dependence Graph* (COSDG), for use as an internal representation for performing test coverage analysis. Apart from representing basic features like control flow, data flow, and method calls, COSDG captures object-oriented features such as class, inheritance, and polymorphism. Novel features of COSDG include details of method visibility in a derived class, and different types of method call edges to depict different calling contexts: simple, inherited, and polymorphic method calls.

Test coverage techniques can be applied to COSDG to obtain the necessary object-oriented coverage measures. A prototype tool has been developed in Java for test coverage analysis of object-oriented programs using the COSDG. We have conducted experimental studies to ascertain the efficacy of COSDG in testing object-oriented programs. Results from our study show that both space and time required to construct the COSDG is linear in program size.

The present version of COSDG represents only the basic object-oriented features. We are extending our representation to include exception handling features.

References

- [1] Hira Agrawal. Efficient coverage testing using global dominator graphs. In *Proc. of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engg. (PASTE '99)*, pages 11–20, Toulouse, France, Sep 1999.
- [2] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, Jul 1970. Proc. of a Symposium on Compiler Optimization.
- [3] ANTLR. ANother Tool for Language Recognition. <http://www.antlr.org/>. Date Accessed: 31 Sep. 2009.
- [4] Peter J. Clarke and Brian A. Malloy. A taxonomy of OO classes to support the mapping of testing techniques to a class. *Journal of Object Technology*, 4(5):95–115, Jul-Aug 2005.
- [5] Eclipse. <http://www.eclipse.org/>. Date Accessed: 31 Sep. 2009.
- [6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, Jul 1987.
- [7] Mary Jean Harrold and Greg Rothermel. A coherent family of analyzable graphical representations for object-oriented software. Tech. Report OSU-CISRC-11/96-TR60, Department of Computer and Information Science, The Ohio State University, Nov 1996.
- [8] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan 1990.

- [9] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. Static slicing of Java programs. Tech. Report TR-96-108, Research Group on Artificial Intelligence, Hungarian Academy of Sciences, József Attila University, Hungary, Dec 1996.
- [10] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proc. of the 18th International Conference on Software Engineering*, pages 495–505, Berlin, Germany, Mar 1996.
- [11] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proc. of the IEEE Intl. Conf. of Software Maintenance (ICSM '98)*, pages 358–367, Bethesda, MD, USA, Nov 1998.
- [12] Raghuram Lingampally, Atul Gupta, and Pankaj Jalote. A multipurpose code coverage tool for Java. In *Proc. of the 40th Annual Hawaii Intl. Conf. on System Sciences (HICSS '07)*, pages 261b – 271b, Jan 2007.
- [13] Brian A. Malloy, John D. McGregor, Anand Krishnaswamy, and Murali Medikonda. An extensible program representation for object-oriented software. *ACM SIGPLAN Notices*, 29(12):38–47, Dec 1994.
- [14] E S F Najumudheen. An intermediate representation for test coverage analysis of object-oriented programs. Tech. Report IITKGP-CSE-TR-17/2008, Indian Institute of Technology, Kharagpur, India, Jan 2008.
- [15] E S F Najumudheen, Rajib Mall, and Debasis Samanta. A dependence graph-based test coverage analysis technique for object-oriented programs. In *Proc. of the 6th Intl. Conf. on Info. Technology: New Generations (ITNG '09)*, pages 763–768, Las Vegas, NV, USA, Apr 2009.
- [16] Karl J. Ottenstein. *Data-Flow Graphs as an Intermediate Program Form*. PhD thesis, Computer Sciences Dept., Purdue Univ., Lafayette, IN, Aug 1978.
- [17] Gregg Rothermel and Mary Jean Harrold. Selecting regression tests for object-oriented software. In *Proc. of the Intl. Conf. on Software Maintenance - 1994*, pages 14–25, Victoria, BC, Canada, Sep 1994.
- [18] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in Java. In *Proc. of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 1–11, Boston, MA, USA, Jul 2004, Vol. 29 No. 4.
- [19] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, Jun 2004.
- [20] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.
- [21] A.M.R. Vincenzi, J.C. Maldonado, W.E. Wong, and M.E. Delamaro. Coverage testing of Java programs and components. *Science of Computer Programming*, 56(1-2):211–230, Apr 2005.
- [22] Neil Walkinshaw, Marc Roper, and Murray Wood. The Java system dependence graph. In *Proc. of the Third IEEE Intl. Workshop on Source Code Analysis and Manipulation, (SCAM '03)*, pages 55–64, Amsterdam, The Netherlands, Sep 2003.
- [23] Jianjun Zhao. Applying program dependence analysis to Java software. In *Proc. of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, Tainan, Taiwan, Dec 1998.

About the authors

E.S.F. Najumudheen is currently pursuing the Ph.D. degree in the Department of Computer Science and Engineering at the Indian Institute of Technology, Kharagpur, India. He received the bachelor's degree in Applied Sciences from Madurai Kamaraj University. He holds two master's degree, one in Computer Applications from Anna University, and another in Computer and Information Technology from IIT, Kharagpur. He has around two years of experience in the software industry, and fourteen years of experience in teaching and research. His current research investigates test coverage analysis, object-oriented testing, and program analysis. He is member of the IEEE, ACM, and ACM SIGSOFT. He can be reached at najum@cse.iitkgp.ernet.in.



Rajib Mall is currently a professor in the Department of Computer Science and Engineering at the Indian Institute of Technology, Kharagpur, India. He has been with IIT, Kharagpur for the past 15 years. He received the bachelor's, master's, and Ph.D. degrees in Computer Science and Engineering from the Indian Institute of Science, Bangalore, India. His research interests include program analysis, program slicing, and software testing. His current research focuses on object-oriented and regression testing. He has published over 100 research papers in refereed journals and conferences, and has authored two books. He is a member of the domain experts board of the International Journal of Patterns(IJOP). He served as the general chair for the IEEE Indicon 2004 and the program chair for CIT 2005 conferences. He has also served as a program committee member for many conferences of international repute. He is a senior member of the IEEE, and has twice served as the chair of the IEEE Kharagpur section. He can be reached at rajib@cse.iitkgp.ernet.in. See also <http://www.facweb.iitkgp.ernet.in/~rajib>.



Debasis Samanta is currently an assistant professor in the School of Information Technology at the Indian Institute of Technology, Kharagpur, India. He received the bachelor's and master's degree in Computer Science and Engineering from Calcutta University and Jadavpur University, respectively, and the Ph.D. degree in Computer Science and Engineering from IIT, Kharagpur. He has more than fifteen years of experience in teaching, and has published more than 50 research papers in refereed journals and conferences. He has also authored two books. He is a senior member of the IEEE, and served as the chair of IEEE Kharagpur Section, India Council, during 2009. He can be reached at dsamanta@sit.iitkgp.ernet.in. See also <http://www.facweb.iitkgp.ernet.in/~dsamanta/>.