An Overview of Slicing Techniques for Object-Oriented Programs

Durga Prasad Mohapatra, Rajib Mall and Rajeev Kumar[1]
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, WB 721 302, India
E-mail: {durga, rajib, rkumar}@cse.iitkgp.ernet.in

Keywords: program slicing, program dependence graph, debugging, object-oriented programs, concurrent object-oriented program, multi-threading, distributed programming.

Received: April 15, 2005

This paper surveys the existing slicing techniques for object-oriented programs. Many commercial object-oriented programs are concurrent in nature. Concurrency is typically implemented in the form of multi-threading or message passing using sockets or both. We therefore review the available techniques in slicing of concurrent object-oriented programs. Another trend that is clearly visible in object-oriented programming is client-server programming in a distributed environment. We briefly review the existing techniques for slicing of distributed object-oriented programs

Povzetek: Opisana je tehnika analize objektnih programov.

1 Introduction

Program slicing is a program analysis technique. The main applications of program slicing includes various software engineering activities such as program understanding, debugging, testing, program maintenance, complexity measurement etc. It can also be used to extract the statements of a program that are relevant to a given computation. A *program slice* consists of the parts or components of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. Typically, a slicing criterion consists of a pair $< s, V >$, where s is the statement number and V is a variable. The components of a program which have a direct or indirect effect on the values computed at a slicing criterion $< s, V >$ are called the *program slice with respect to the slicing criterion* $< s, V >$.

The concept of a program slice was introduced by Weiser [92]. Various slightly different notions of program slices have been proposed. There has also been a proliferation of the number of methods to compute slices. The main reason for this proliferation of slicing techniques is that different applications require different properties of slices. Weiser [91] defined a program slice S as a *reduced, executable program* obtained from a program P by removing statements, such that S replicates part of the behavior of P. The program slicing technique originally introduced by Weiser [91, 92, 93] is now called *static* backward slicing. It is static in the sense that the slice is independent of the input values to the program. It is *backward* because the control flow of the program is considered in reverse while constructing the slice. Another common definition of a slice is a subset of the statements and control predicates of the program which directly or indirectly affect the values computed at the slicing criterion, but which do not necessarily constitute an executable program.

Object-oriented programming style is becoming the norm. These programs may be very large as well as concurrent. In some applications the programs run in a distributed manner on several nodes connected through a network. The code size of these object-oriented programs often exceeds million of lines. Making such programs dependable and trust worthy is a major challenge. Program slicing is advocated as a technique to automatically analyze a program. The results of the analysis can be used to help in debugging, test case design, test coverage analysis and others [94, 82, 96, 19].

Slicing object-oriented programs presents new challenges which are not encountered in traditional program slicing. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance, message passing and polymorphism need to be considered carefully. Although the concepts of inheritance and polymorphism are strengths of object-oriented programming languages, they pose special challenges in program slicing. Due to *inheritance* and *dynamic binding* in object-oriented programs, the process of tracing dependencies becomes more complex than that in a procedural program. Larson and Harrold were the first to consider these aspects in their work [60]. To address these object-oriented features, they enhanced the system dependence graphs (SDG) [48] to represent object-oriented software. After the SDG is constructed, the two phase algorithm of Horwitz et al. [48] is used with minor modifications for computing static slices. Larson and Harrold have reported only a static slicing technique for object-oriented programs [60], and did not address dynamic slicing aspects. The dynamic slicing aspects have been reported by Zhao [100], Song et al. [84], Xu et

al. [94] and Wang et al. [90].

Most of the commercial object-oriented programs are concurrent in nature and run in different machines connected through a network. It is usually accepted that understanding and debugging *concurrent* and *distributed* object-oriented programs are much harder compared to the sequential programs. Slicing techniques promise to come in handy at this point. However, most of the research work in the program slicing area have focused attention on sequential programs. Research reports addressing slicing of *concurrent* and *distributed* object-oriented programs are scarce in literature [102, 17, 54, 55, 75].

Several comprehensive surveys are available for program slicing in general [86, 12, 63, 45, 95, 20]. But, surveys on slicing of object-oriented programs have not been reported to the literature to the best of our knowledge. In this paper, we present a brief survey of the existing slicing techniques for object-oriented programs. Also, we have reviewed the available literatures on the available techniques for slicing concurrent object-oriented programs. Subsequently, we have discussed the current trend in the area of slicing of distributed object-oriented programs. In this survey, we discuss the contribution of each work and compare the major difference between them.

In the following, we review some basic slicing concepts that would be useful to understand the rest of the paper.

1.1 Categories of Program Slicing

Several categories of program slicing as well as methods to compute them are found in literature. The main reason for the existence of so many categories of slicing is the fact that different applications require different types of slices. Slices can be *backward* or *forward* [48, 98], *static* or *dynamic* [3, 52, 27], *intra-procedural* or *inter-procedural* [48].

Static Slicing and Dynamic Slicing: *Static slicing* technique uses static analysis to derive slices. That is, the source code of the program is analyzed and the slices are computed for *all possible input values*. Therefore static slices are conservative and contain more statements than necessary. For object-oriented programs the situation is still worse as the computed static slice will contain all most all of the statements present in the program. This is due to the various relationships such as inheritance, polymorphism, dynamic binding etc. existing among classes. Therefore, static slices are of little use in the context of object-oriented programs.

Korel and Laski [52] introduced the concept of dynamic program slicing. Dynamic slicing makes use of the information about a *particular execution* of a program. *A dynamic slice with respect to a slicing criterion* $< s, V >$*, for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution*. Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications

```
1 main()
2 {
3   int i, sum;
4    cin>> i;
5    sum = 0;
6    while(i <= 10)
7    {
8      sum=sum+i;
9       ++ i;
10   }
11   cout<< sum;
12   cout<< i;
13 }
```

Figure 1: An example program

such as program debugging and testing. *Dynamic slicing* is more suitable for object-oriented programs than *static slicing* as the computed dynamic slice will contain only those statements that *actually* affect the slicing criterion. In other words, we can say that dynamic slicing techniques compute *precise slices*. A comprehensive survey on the existing dynamic program slicing algorithms is reported in Korel and Rilling [53] and Xu et al. [95].

Consider the C++ example program given in Fig. 1. The static slice with respect to the slicing criterion $< 11, sum >$ is the set of statements {4, 5, 6, 8, 9}. Consider a particular execution of the program with the input value $i = 15$. The dynamic slice with respect to the slicing criterion $< 11, sum >$ for the particular execution of the program is {5}.

Backward Slicing and Forward Slicing: As already discussed, a backward slice contains all parts of the program that might directly or indirectly affect the slicing criterion [92]. Thus a static backward slice provides the answer to the question: *"which statements affect the slicing criterion?"*.

A forward slice with respect to a slicing criterion $< s, V >$ contains all parts of the program that might be affected by the variables in V used or defined at the program point s [84, 98]. A forward slice provides the answer to the question: *"which statements will be affected by the slicing criterion?"*.

Intra-procedural Slicing and Inter-procedural Slicing: Intra-procedural slicing computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, inter-procedural slicing can be used to derive slices that span multiple procedures [48].

For object-oriented programs, intra-procedural slicing is meaning less as practical object-oriented programs contain more than one method. So, for object-oriented programs, inter-procedural slicing is more useful.

Other Slicing Categories: Many examples of slicing are combinations of the categories above. For example, Weiser's original work [91] describes backward, static, intra-procedural slicing; although he also later gave an algorithm for backward, static, inter-procedural slicing [93]. The work of Kamkar [49] produces backward, dynamic, inter-procedural slicing. It is also possible to combine the features of static slicing with the features of dynamic slicing. This new form of slicing is called hybrid slicing [40]. Hybrid slicing is an approach for refining static slices using dynamic information.

There are variants of slicing in between the two extremes of static and dynamic, where some but not all properties of the initial state are known. These are known as *conditioned slices* [13, 31, 42, 25] or *constrained slices* [29]. Traditional slicing methods are all based on statement deletion. In a recently reported form of slicing called *amorphous slicing* [11, 43, 46], slices are not necessarily produced by deleting statements and may not necessarily even be made from components of the original program being sliced. The slice is computed based on the *semantics* of the program. Recently, another form of slicing called *modular monadic slicing* has been developed where slices are computed based on the modular monadic semantics of the program analyzed [99]. This method computes slices directly on abstract syntax of the program without constructing intermediate representations such as dependence graphs.

1.2 Applications of Program Slicing

This section describes the use of program slicing techniques in various applications. In trying to use the basic slicing concepts in diverse domains, several variations of the notions of program slicing as described in Section 1 are developed. The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging [92, 64]. From this modest beginning, the use of program slicing techniques has now ramified into a powerful set of tools for use in such diverse applications as program understanding, program verification, automated computation of several software engineering metrics, software maintenance and testing, functional cohesion, dead code elimination, reverse engineering, parallelization of sequential programs, software portability, reusable component generation, compiler optimization, program integration, showing differences between programs, software quality assurance, software fault-injection etc. [10, 96, 65, 86, 32, 49, 44, 39, 104, 21, 30, 42, 19]. Slicing methods also play an important role in software fault-injection, see [89] for using slicing methods in software fault-injection. A comprehensive study on the applications of program slicing is made by Binkley and Gallagher [12], Lucia [63] and Qi et al. [82].

1.3 Paper Organization

The remainder of this paper is organized as follows. In Section 2, we discuss the inter-procedural slicing technique, that would be useful in understanding slicing of object-oriented programs. In Section 3, methods for slicing object-oriented programs are discussed. In Section 4, we review the slicing techniques for concurrent object-oriented programs. In Section 5, techniques for slicing of distributed object-oriented programs are discussed. Section 6 concludes the paper.

2 Inter-Procedural Slicing

Horwitz et al. [48] developed the *system dependence graph* (SDG) as an intermediate program representation and proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. A system dependence graph is a collection of procedure dependence graphs, one for each procedure. A *procedure dependence graph* represents a procedure as a graph in which vertices are statements or predicate expressions and the edges represent the dependence relationships. There are two types of dependence edges: data dependence edge and control dependence edge. *Data dependence* edges represent flow of data between statements or expressions, and *control dependence* edges represent control conditions on which the execution of a statement or expression depends. Each procedure dependence graph contains an *entry vertex* that represents entry into the procedure. To model parameter passing, an SDG associates each procedure entry vertex with *formal-in* and *formal-out* vertices. An SDG contains a formal-in vertex for each formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified by the procedure. An SDG associates each call site in a procedure with a *call* vertex and a set of *actual-in* and *actual-out* vertices. An SDG contains an actual-in vertex for each actual parameter at the call site and an actual-out vertex for each actual parameter that may be modified by the called procedure. At procedure entry and call sites, global variables are treated as parameters. Thus, there are actual-in, actual-out, formal-in and formal-out vertices for these global variables. SDGs connect procedure dependence graphs at call sites. A *call* edge connects a procedure call vertex to the entry vertex of the called procedure's dependence graph. Parameter-in and parameter-out edges represent parameter passing. *parameter-in* edges connect actual-in and formal-in vertices, and *parameter-out* edges connect formal-out and actual-out vertices. Horwitz et al. compute inter-procedural slices by solving a graph reachability problem on the SDG. To obtain precise slices, the computation of a slice must preserve the calling context of called procedures, and ensure that only paths corresponding to legal $call/return$ sequences are considered. To facilitate the computation of inter-procedural slicing that considers the calling context, an SDG represents the flow of dependencies across call sites. A *transitive flow*

of dependence occurs between the actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence may be caused by data dependencies, control dependencies or both. A *summary edge* models the transitive flow of dependence across a procedure call. Fig. 2 represents a simple example program containing two procedures i.e. *add* and *inc*. The system dependence graph of Fig. 2 is shown in Fig. 3. In the SDG of Fig. 3, circles represent program statements and ellipses represent parameter vertices.

After constructing the SDG, Horwitz et al. [48] applied a two-pass algorithm on the SDG to compute the static slices. The first pass of the inter-procedural slicing algorithm traverses backward along all edges except parameter-out edges, and marks those vertices reached. The second pass traverses backward from all vertices marked during the first pass along all edges except call and parameter-in edges, and marks reached vertices. The slice is union of the vertices marked during pass one and pass two.

3 Slicing of Object-Oriented Programs

In this section, we first discuss some work on static slicing of object-oriented programs. Then, we discuss how these basic slicing techniques have subsequently been extended by researchers to handle dynamic slicing of object-oriented programs.

3.1 Static Slicing of Object-Oriented Programs

Static slicing of object-oriented programs has drawn considerable research interest [56, 60, 88, 87, 62, 16, 59, 57, 58, 47, 66, 41]. While slicing object-oriented programs, how to represent the programs is an important problem. Larson and Harrold [60] extended the SDG of Horwitz et al. [48] to represent object-oriented programs. They have constructed Class Dependence Graphs (ClDG) for each class in an object-oriented program. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments. Each method in a ClDG is represented by a procedure dependence graph [60]. Each method has a *method entry* vertex that represents the entry into the method. A ClDG also contains a *class entry* vertex that is connected to the method entry vertex for each method in the class by a *class member* edge. Class entry vertices and class member edges let us quickly access method information when a class is combined with another class or system. The ClDG construction expands each method entry by adding formal-in and formal-out vertices similarly as procedure dependence graphs.

Fig. 4 contains an example program written in C++ which creates the class *Elevator* and *AlarmElevator* depending on the command line arguments. Fig. 5 shows the ClDG for the *Elevator* class. A rectangle represents the class entry vertex and circles represent the statements. The ellipses represent the parameter vertices. For example in Fig. 5, the vertex 1 is the class entry vertex and 2, 6, 7, 9, 11, 13, 15 and 21 are method entry vertices. Bold dashed edges represent class member edges that connect class entry vertex to method entry vertex. For example (1, 2), (1, 6), (1, 7), (1, 9) and (1, 11) are class member edges. Each method entry vertex is the root of a subgraph that is itself a partial SDG containing control dependence edges, data dependence edges, call and parameter edges, and summary edges.

Since methods in a class can interact with each other or with other methods, a ClDG represents the effects of method calls by a *call* vertex. At each call vertex, there are actual-in and actual-out vertices to match the formal-in and formal-out vertices present at the entry to the called method. For example, in Fig. 5, vertices 18 and 20 represent calls to *add*().

To represent *derived class*, Larson and Harrold constructed a ClDG for the derived class by constructing a representation for each method defined by the derived class, and reusing the representations of all methods that are inherited from the base classes [60].

A *polymorphic method call* occurs when a method call is made and the destination of the call is unknown at compile time. The ClDG should represent the polymorphic method call. For this purpose, the ClDG uses a *polymorphic choice vertex* to represent the dynamic choice among the possible destinations. A call vertex corresponding to a polymorphic call has a call edge incident to a polymorphic choice vertex. A polymorphic choice vertex has call edges incident to subgraphs that represent calls to each possible destination. The polymorphic choice vertex represents the dynamic selection of a destination. In fig. 6 $P1$ is a polymorphic choice vertex that represents a dynamic choice between calls to *Elevator::go*() and *AlarmElevator::go*(). The two unlabeled vertices associated with *P1* represent the dummy polymorphic choice vertices.

At last, the SDG for a complete program is constructed by connecting calls in the partial system dependence graph to methods in the ClDG for each class. This process involves connecting call vertices to method entry vertices, actual-in vertices to formal-in vertices, and formal-out vertices to actual-out vertices. The summary edges for methods in a previously analyzed class are added between the actual-in and actual-out vertices at call sites. This construction of the SDG for an object-oriented system maximizes reuse of previously constructed portions of the representation.

Fig. 4 contains an example of an application program that instantiates an object. The SDG of Fig. 4 is given in Fig. 6. The variable *e_ptr* could point to an object of type *Elevator* or *AlarmElevator*. This graph was constructed by building a partial SDG for the *main* function, including the previously computed representation for the *Elevator* and

```
main( )
 int  s, i;
{
  s = 0;                      void  add(int a, int b)        void inc(int z)
  i = 1;                            {                              {
  while (i  < 10) do                  a = a + b;                     add(z,1);
     {                                return;                        return;
         add(s, i);                  }                              }
          inc(i);
      }
  write(s);
 }
```

Figure 2: An example program

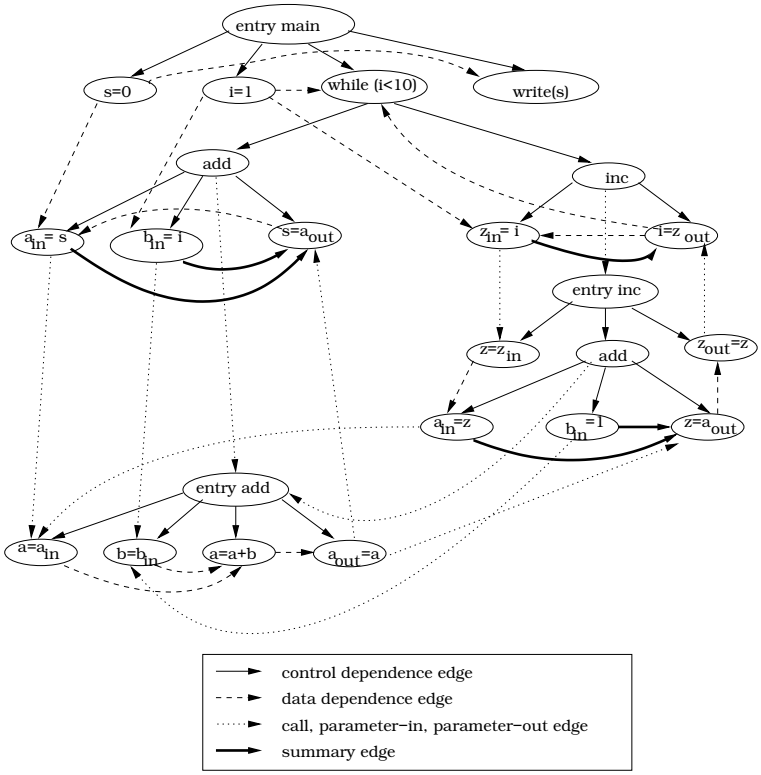

Figure 3: The system dependence graph of the example program of Fig. 2

```
1:  class Elevator{
        public:
2:      Elevator(int 1_top_floor)    /* initialization for Elevator */
3:        { current_floor = 1;
4:          current_direction = UP;
5:          top_floor = 1_top_floor; }     /* end of Elevator */
6:      virtual  ~Elevator() { }
7:      void up()
8:        { current_direction = UP; }
9:      void down()
10:       {  current_direction = DOWN;  }
11:     int which_floor()
12:       { return current_floor;  }
13:     Direction direction()
14:       { return current_direction;  }
15:     virtual void go(int floor)   /* declaration for method go() */
16:       { if  (current_direction = UP )
17:             { while (current_floor != floor)
                      &&  (current_floor <= top_floor)
18:               add(current_floor, 1); }
                  else
19:               { while (current_floor != floor)
                        && (current_floor  > 0 )
20:               add(current_floor, −1);  } /* end if */
            };
          private:
21:       add(int &a, const int &b) /* This method computes value of current_floor */
22:           {  a = a+b; } ;
          protected:
            int current_floor;
            Direction current_direction;
            int top_floor;
          };
23:     class AlarmElevator: public Elevator {  /* AlarmElevator is derived from  Elevato
          public:
24:       AlarmElevator(int top_floor);
25:           Elevator(top_floor)
26:               {alarm_on = 0;   }
27:       void set_alarm()
28:               {alarm_on = 1;  }
29:       void reset_alarm()
30:               {alarm_on = 0;  }
31:       void go(int floor)
32:           { if (! alarm_on)
33:             Elevator :: go(floor);
                };
          protected:
            int alarm_on;
          };
34: main(int argc, char  **argv)  {
          Elevator *e_ptr;
35:     if (argv[1])
36:           e_ptr = new Elevator(10);
          else
37:           e_ptr = new AlarmElevator(10);
38:     e_ptr −> go(3);      /* polymorphic method call */
39:     cout << "\n currently on floor:"
                << e_ptr −> which_floor();
        } /* end of main */
```

Figure 4: An example program

AlarmElevator classes, and connecting each graph using call, parameter-in, and parameter-out edges. In Fig. 6, the left hand side keys represent keys for formal parameter vertices and right hand side keys represent keys for actual parameter vertices.

After constructing the SDG for a complete object-oriented program, they have used the two-pass graph reachability algorithm [48] for computing slices. Fig. 6 shows the SDG of the example program given in Fig. 4 and the static slice with respect to the call to *which_floor()* at vertex 39, which includes all statements that may affect *current_floor*. The shaded vertices in the SDG represent the statements included in the slice. The static slice is shown in Fig. 7 in more detail. Since Larson and Harrold [60] have

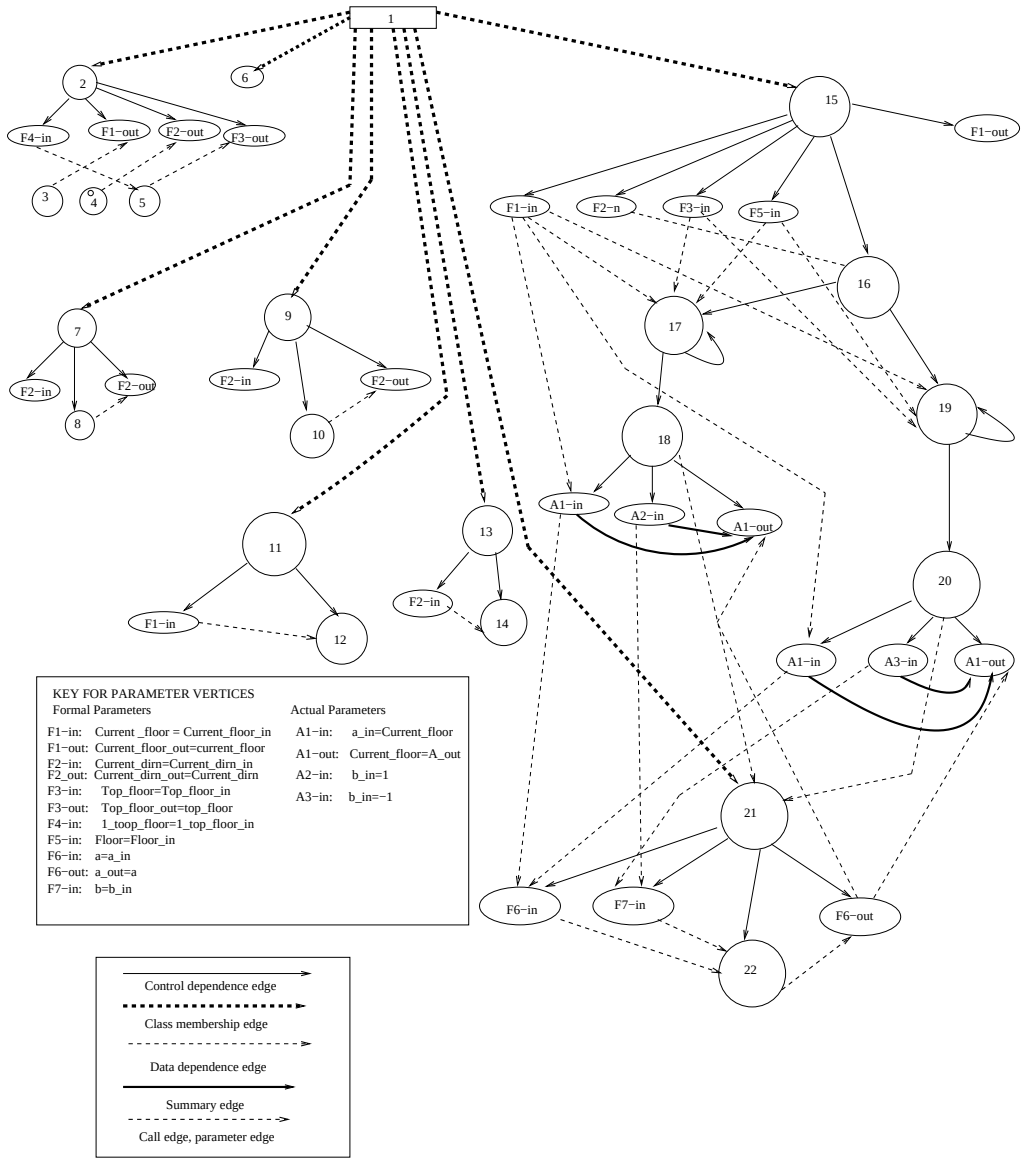

Figure 5: The ClDG for class *Elevator*

computed the static slice, so all most all of the statements in the example program are included in the slice.

One limitation of this approach is that the data dependencies obtained using the approach for creating the individual procedure dependence graphs are imprecise: by treating data members declared in a class as if they were global to the methods of that class, the approach fails to consider the fact that in different method invocations, the data members used by the methods might belong to different objects. A second limitation of the approach is that it does not handle cases in which an object is used as a parameter or as a data member of another object.

Tonella et al. [88] have addressed the first limitation by extending a methods signature to include data members of the class as formal parameters so that an object can pass its data members into the method as actual parameters. Their approach, however, is unnecessarily expensive because each method call site has actual parameter vertices for all data members of the object, even if only a few of them are referenced by the method. They addressed the second limitation by representing an object as a single vertex when the object is used as a parameter. This representation, however, might cause the slicer to produce imprecise slices because the slice may include all the data members of the object even if a few of them affects the slicing criterion.

Liang et al. [62] developed a more efficient intermediate representation to overcome the above limitations. To obtain more precision when an object is used as a parameter (*parameter object*), their modified SDG explicitly represents the data members of the object. They have represented the parameter object as a *tree*. The root of the tree represents the object itself, the children of the root represent the data members of the object and the edges of the tree represent the data dependencies between the object and it's

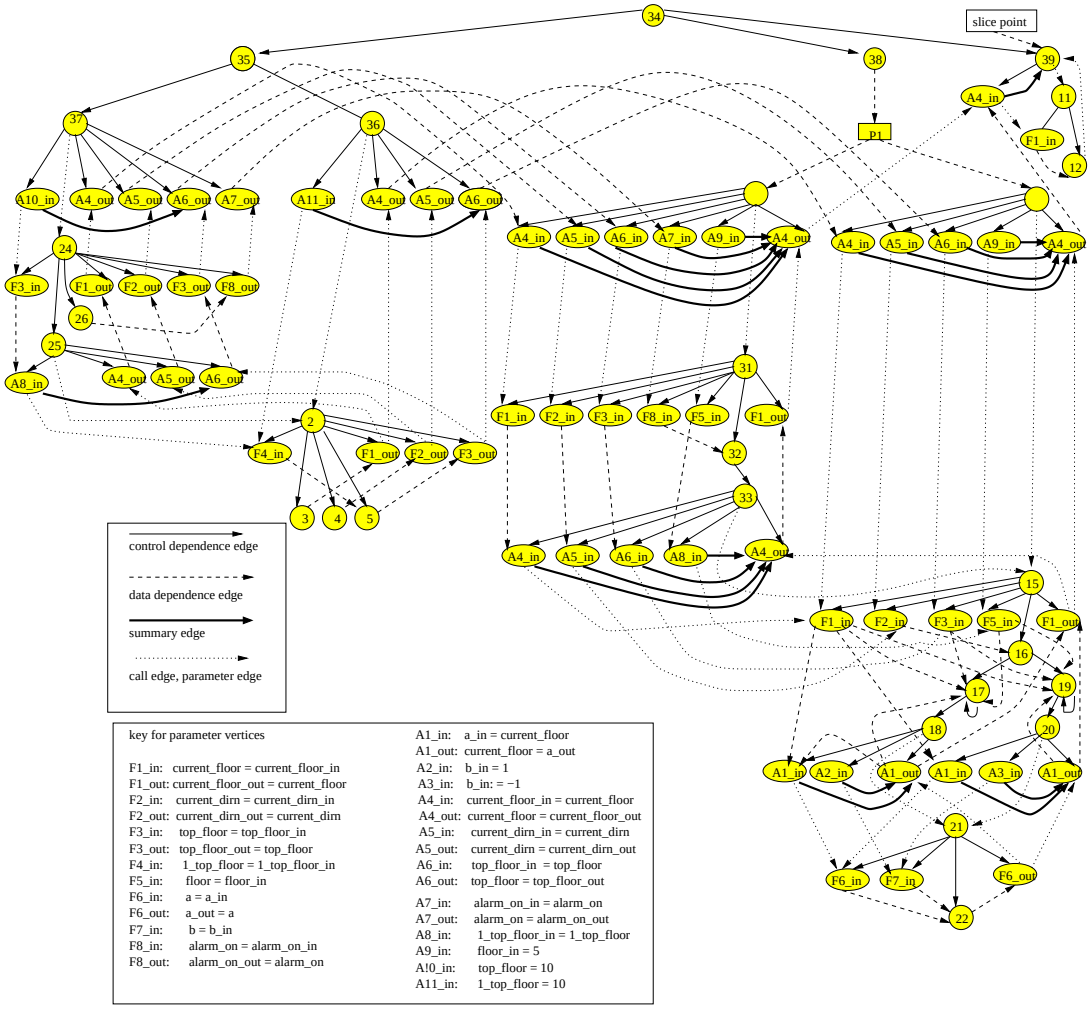

Figure 6: The system dependence graph of Fig. 4

data members. Under this representation, if a data member of the object is another object, then this data member can be further expanded into a subtree.

In this representation, a *polymorphic object* is represented as a tree in which the root of the tree represents the polymorphic object itself and the children of the root represent objects of the possible types. When the polymorphic object is used as a parameter, the children are further expanded into trees. When the polymorphic object receives a message, the children are further expanded into call sites. Note that, in this case, the technique of Liang et al. [62] differs from that of Larson and Harrold [60]. Liang et al. have used one call site for each possible object type, in their representation. But in the representation of Larson and Harrold, different call sites are used only for different implementations of a virtual method.

To represent inheritance, Liang et al. [62] have maintained one copy of the representation for a method within a class hierarchy. Then, this representation can be shared by different classes in the hierarchy. The class entry vertex in the SDG groups the methods belonging to one class together using *class member edges* [60]. But in some cases, a method might require a new representation when the program dependence graph for a new class to the hierarchy, is constructed. Liang et al. [62] suggested that, a method will require a new representation, if

- the method is declared in the new class, or
- the method is declared in a lower level class in the hierarchy and calls a newly redefined virtual method directly or indirectly.

Liang et al. [62] have also introduced a new concept called *object slicing*, which enables the user to inspect the effects of a particular object on the slicing criterion. Object slicing provides better support for debugging and program understanding for large scale programs. Sometimes the user may like to focus attention on one object at a time. To do this, they have designed a method to identify the statements in the methods of a particular object that might affect the slicing criterion.

The shortcomings of their method are that:

1. When slicing the object, we must obtain the complete

```
1:  class Elevator{
        public:
2:      Elevator(int 1_top_floor)          /* initialization for Elevator */
3:        { current_floor = 1;
4:          current_direction = UP;
5:          top_floor = 1_top_floor; }              /* end of Elevator */
6:        virtual  ~Elevator() { }
7:        void up()
8:           { current_direction = UP; }
9:        void down()
10:          {  current_direction = DOWN;  }
11:     int which_floor()
12:       { return current_floor;  }
13:       Direction direction()
14:          { return current_direction;  }
15:     virtual void go(int floor)          /* declaration for method go() */
16:       { if  (current_direction = UP )
17:           { while (current_floor != floor)
                  &&  (current_floor <= top_floor)
18:           add(current_floor, 1); }
              else
19:           { while (current_floor != floor)
                  && (current_floor  > 0 )
20:           add(current_floor, −1);  }                    /* end if */
            };
          private:
21:       add(int &a, const int &b)          /* This  method computes value of current_floor */
22:         {  a = a+b; } ;
          protected:
            int current_floor;
            Direction current_direction;
             int top_floor;
           };
23:     class AlarmElevator: public Elevator    /* AlarmElevator is derived from  Elevator */
          public:
24:       AlarmElevator(int top_floor);
25:         Elevator(top_floor)
26:           {alarm_on = 0;   }
27:         void set_alarm()
28:               {alarm_on = 1;  }
29:         void reset_alarm()
30:               {alarm_on = 0  }
31:       void go(int floor)
32:         { if (! alarm_on)
33:           Elevator :: go(floor)
                };
            protected:
                int alarm_on;
            };
34: main(int argc, char  **argv)  {
          Elevator *e_ptr;
35:     if (argv[1])
36:         e_ptr = new Elevator(10);
          else
37:         e_ptr = new AlarmElevator(10);
38:     e_ptr −> go(3);          /* polymorphic method call */
39:     cout << "\n currently on floor:"
                << e_ptr −> which_floor();
          }/* end of main */
```

Figure 7: The static slice of Fig. 4 on slicing criterion (39, current_floor)

slice first for the program. This might be too expensive.

2. When an object's method invokes other methods or is invoked by other methods, we must traverse backward through several methods.

Hammer et al. [41] proposed a new slicing algorithm for Java, which includes all dependencies between fields of nested objects but is more precise than previous algorithms [60, 88, 62]. Instead of limiting the tree level, Hammer et al. [41] unfold the tree completely. As this is not possible for recursive data structures, they have presented a *condition for safe termination of unfolding*. The condition is based on points-to information. This method keeps all trees finite but guarantees that no dependencies are lost. Points-to information is also used to constrain run-time targets of method calls. As a by-product, a call graph is extracted. But, even the best points-to analysis will not resolve all object polymorphism, and the object trees must represent all possible run-time types of an object. Unlike [62], Hammer et al. [41] do not represent polymorphic objects as a set of trees, but as one *merged tree*. To disambiguate fields with the same name but defined in different classes, they have used the fully qualified field name. Thus merging does not reduce the precision of the final SDG. It just reduces the size of the SDG. The short coming of this

approach is that it is more expensive than [60, 62].

Krishnaswamy [56] proposed a different approach to slicing object-oriented programs. He used another dependence-based representation called the *object-oriented program dependency graph* (OPDG) to represent the object-oriented programs. The OPDG of an object-oriented program represents control flow, data dependencies and control dependencies. The OPDG representation of an object-oriented program is constructed in three layers, namely: *Class Hierarchy Subgraph* (CHS), *Control Dependence Subgraph* (CDS), and *Data Dependence Subgraph* (DDS). The CHS represents inheritance relationship between classes, and the composition of methods into a class. A CHS contains a single *class header node* and a *method header node* for each method that is defined in the class. Inheritance relationships are represented by edges connecting class headers. Every method header is connected to the class header by a *membership edge*. Subclass representations do not repeat representations of methods that are already defined in the super classes. *Inheritance edges* of a CHS connect the class header node of a derived class to the class header nodes of it's super classes. *Inherited membership edges* connect the class header node of the derived class to the method header nodes of the methods that it inherits. A CDS represents the static control dependence relationships that exists within and among the different methods of a class. The DDS represents the data dependence relationship among the statements and predicates of the program. The OPDG of an object-oriented program is the union of three subgraphs: CHS, CDS and DDS. Slices can be computed using OPDG as a graph-reachability problem. He also computed the polymorphic slices of object-oriented programs based on the OPDG.

The OPDG of an object-oriented program is constructed as the classes are compiled and hence it captures the complete class representations. The main advantage of OPDG representation over other representations is that the representation has to be generated only once during the entire life of the class. It does not need to be changed as long as the class definition remains unchanged. Fig. 8 represents the CHS of the example program of Fig. 4.

Kung et al. [59, 57, 58] presented a representation for object-oriented software. Their model consists of an *object relation diagram* and a *block branch diagram*. The object relation diagram of an object-oriented program provides static structural information on the relationships existing between objects. It models the relationship that exists between classes such as inheritance, aggregation and association. The block branch diagram of an object-oriented program contains the control flow graph of each of the class methods, and presents a static implementation view of the program. Harrold and Rothermel [47] presented the concept of *Call Graph*. A call graph provides a static view of the relationship between object classes. A call graph is an inter-procedural program representation in which nodes represent individual methods and edges represent call sites. However, a call graph does not represent important object-oriented concepts such as inheritance, polymorphism and dynamic binding.

Chen et al. [14] proposed an intermediate representation called *Object-Oriented Dependency Graph* (ODG) to represent object-oriented programs. The ODG is a multi-diagraph which is extended from a directed graph by augmenting multiple edge types, vertex properties, and property relations. With this extension, the ODG can avoid some dependencies due to object encapsulation. Based on the ODG, Chen et al. [14] presented an algorithm for slicing of object-oriented programs.

Chen et al. [15] defined two types of program slices, state and behavior slices by considering the dependencies of object-oriented features. A state slice for an object is a set of messages and control statements that might affect the state of the object. A behavior slice for an object is a set of attributes and methods defined in related classes that might affect the behavior of the object.

Although, these approaches [56, 60, 88, 87, 62, 16, 59, 57, 58, 47, 66, 41] represent many features of object-oriented programs, still there are some drawbacks with these approaches. First, these techniques are not fit to represent larger programs, because all the procedure dependence graphs of subprograms are connected in the SDG, and for a large program the SDG will be too large to manage and understand. Second, the existing techniques only slice statements in methods of a class. A class consists of a set of methods and data members. Statement slicing is not enough to analyze and understand classes. Finally, to improve the efficiency, most of the PDGs of methods should be reused.

To overcome these drawbacks, Chen and Xu [18] have proposed a new approach to represent dependence for object-oriented Java software that is quite different from the existing SDG representations [60, 88, 62, 56], which connect all PDGs of methods. This new program dependence graph is a set of PDGs with tags that are not connected. The PDG of a class consists of a set of PDGs of its methods. Each PDG is an independent graph, and does not connect to any other PDGs. The *tags* have the form (x, y) (where x and y are variables) and are used to distinguish the different definitions and dependencies in a statement. They have used the following sets in their approach: *def(s)*, *ref(s)*, *Def(s, x)*, *Dep_D(s, x)*, and *Dep_R(s)*. They have defined these sets as follows:

- Def(s) denotes the variables whose values are defined (modified) at s. The *in* formal parameters are defined at entry node of the subprogram.
- Ref(s) denotes the variables whose values are refereed, but not modified at s.
- Def(s, x) denotes the variables used when defining variable x at s.
- Dep_D(s, x) = $\{(x, s^1, y)$, such that $y \in Def(s, x) and y \in Def(s^1)$ and there exists a path from s^1 to s on which y is not redefined$\}$.

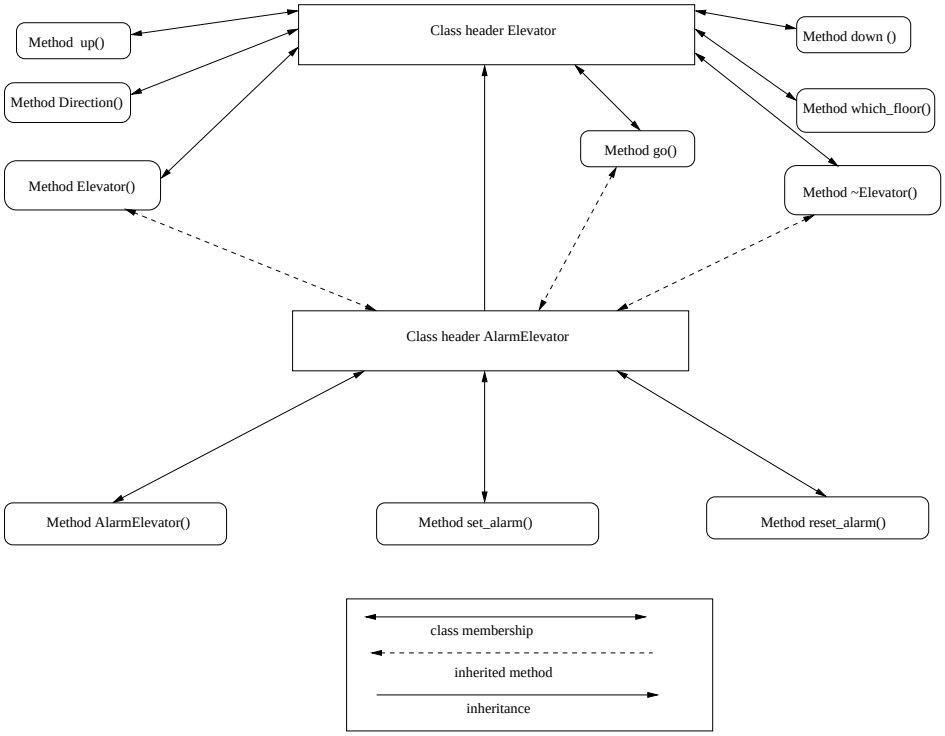

Figure 8: The CHS for the example program of Fig. 4

– Dep_R(s) = $\{(x, s^1, x)$, such that if s is a control statement, x is a conditional variable used at s, and $x \in Def(s^1)$ and there exists a path from s^1 to s on which x is not redefined$\}$.

Chen and Xu [18] defined the program dependence graph (PDG) of a method M as a directed graph with tags. According to this approach, PDG is triplet $< S^1, E^1, T >$, where node set $S^1 = S$ (S is the node set of M's CFG), edge set $E^1 = E_1 \cup E_2$, whereE_1 is the set of direct control dependence edges and $E_2 = \{< s_1, s_2 > suchthat(x, s_2, y) \in Dep\_D(s_1, x) and (x, s_2, x) \in Dep\_R(s_1)\}$ is the set of direct data dependence edges. T is a tag set. The tag on an edge $< s_1, s_2 >$ can be obtained in the following way:

– If $< s_1, s_2 > \in E_1$, then its tag is (*, *);

– If $< x, s_2, y > \in Dep\_D(s_1)$, then its tag is (y, x);

– If $< x, s_2, x > \in Dep\_R(s_1)$, then its tag is (x, x);

For example, consider the sample program in Fig. 4. The PDG of the method *Elevator*, according to the approach of Chen and Xu [18], is shown in Fig. 9. Similarly the PDG of other methods can be drawn. According to the approach of Chen and Xu [18], in Fig. 9, the tag for the edge (2, 3) is (current_floor, 1), the tag for edge (2, 4) is (current_direction, UP) and the tag for the edge (2, 5) is (top_floor, 1). There are three classes, *Elevator*, *AlarmElevator* and *main*, in the sample program. The PDG of class *Elevator* is shown in Fig. 10. It may be observed that the PDG of each method in Fig. 10,is independent. Using the PDG of a method, Chen and Xu [18] solved intra-method slicing as a graph-reachability problem with tags. The approach of Chen and Xu [18] differs from the previous approaches [48] in that it checks not only the edges but also the tags on these edges. Based on this new model, they have introduced the concepts of *partial slicing*, *object slicing* and *class slicing*.

Partial slicing: partial slicing can make the user pay attention to the interesting parts of the program, and slice incomplete programs or components from a third party without source codes. Informally, given a slicing criterion $< s, v >$, the partial slicing only slices the interested parts of the program such as a class, few methods of a class or an object. To slice parts of a program, they have constructed the PDGs of interested subunits. For other methods, only the interfaces, i.e., the dependencies among parameters are needed. It is enough to know the interface (how to use the method) of incomplete programs or components from a third party. When we construct the PDG of a program, all PDGs of methods have been constructed. Based on these PDGs, we can use the partial slicing algorithm. In the slicing algorithm, each method is sliced independently. If the method is not considered, we just do not slice it.

Object slicing: Object slicing was first introduced by Liang et al. [62]. It is mainly used for tasks, such as debugging and program understanding. Object slicing identifies statements in methods of an object that might affect the slicing criterion. To slice an object, the slicing criterion is changed to $< s, v, Object >$. Informally, given a slicing criterion $< s, v, Object >$, object slicing identifies the

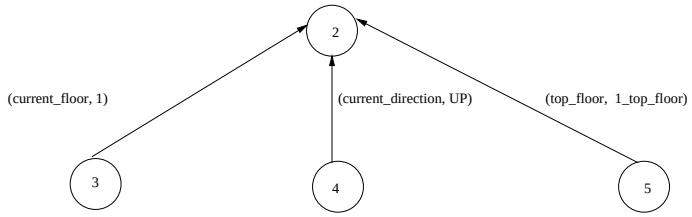

Figure 9: The PDG of method *Elevator* of the example program of Fig. 4

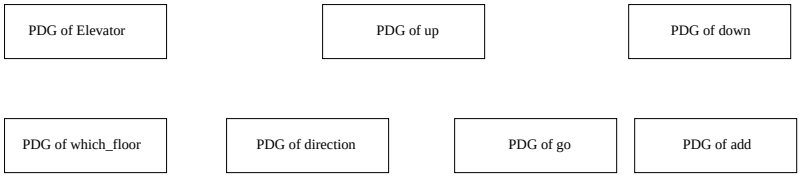

Figure 10: The PDG of class *Elevator* of the example program of Fig. 4

statements in the methods of the object that might affect slicing criterion $< s, v >$.

Class slicing: Class slicing identifies data members and statements in methods of the class that might affect the slicing criterion. Class slicing slices not only the methods, but also all the data members. To slice a class, the slicing criterion is changed to $< s, v, Class >$. Informally, given a slicing criterion $< s, v, Class >$, the result of class slicing of a class is a class that includes partial data members and statements in the methods of class, and these data members and statements might influence the variable defined at s. To slice a class, one method is to union all the object slices of the class and record the data members used. When the number of objects is large, this method will be too expensive. Another way is that, when constructing the PDG, we do not distinguish data members for different objects instantiated from the same class. But using such PDG will lose much information that is useful for other slicing. The best way is to traverse backward from s when slicing and mark the statements and data members used in the class based on this new PDG.

The advantages of this approach [18] are that:

- It distinguishes data members for different objects and represents the effects of polymorphism and dynamic binding.
- Using this representation, the PDGs can be constructed concurrently as each PDG is independent. So, this representation is quite fit for representing larger programs.
- Object slicing enables users to inspect statements in a slice, object by object. Class slicing enables users to inspect not only the statements in methods but also data members in classes.
- According to this slicing algorithm, when the slicing criterion changes, most PDGs need not be traversed, because the previous results that are saved on disks, can be reused.

The shortcoming of this approach is that when we only slice once or few times, the cost might be too much, because all the methods are analyzed first before slicing and the results are stored in libraries on disk.

Steindl [85] has developed a fully operational program slicing tool, *Oberon Slicing Tool*, for the programming language *Oberon-2*. It generates state-of-the-art algorithms and applies them to a strongly-typed object-oriented programming language. It extends them to support intermodular slicing of object-oriented programs. Control and data flow analysis considers inheritance, dynamic binding and polymorphism, as well as side-effects of functions, short circuit evaluation of Boolean expressions and aliases due to reference parameters and pointers. The algorithm for alias analysis is fast but effective by taking into account information about the type of variables and the place of their declaration. The result of static program analysis is visualized with active text elements: hypertext links connect the call sites with the possible call destinations, parameter information elements indicate the direction of data flow at calls. Since static program analysis must make conservative assumptions about actual program executions, the sets of possible aliases and call destinations due to dynamic binding are more general than necessary. Steindl has visualized these sets and allowed the programmer to restrict them via user interaction. These restrictions are then used to compute more precise control and data flow information. In this way, the programmer can limit the effects of aliases and dynamic binding and bring in his knowledge about the program into the analysis.

The disadvantages of this technique are:

- The layout of the the original source code is lost.
- The front-end of the compiler skips all comments, so they are lost and cannot be displayed.
- The front-end of the compiler performs some simple optimizations such as constant folding, transformation

of IF statements with constant conditions, replacement of integer multiplication by a power of two by arithmetic shifts, etc. These optimizations cannot be undone and the results are presented to the user. This may give insights, but may also confuse.

– The reconstruction of the source code is difficult, the module implementing the reconstruction and the user interface is very big, approximately 3000 lines.

3.2 Dynamic Slicing of Object-oriented Programs

Korel and Laski [52] introduced a new form of slicing. This new form of slicing is dependent on input data and is generated during execution-time analysis as opposed to Weiser's static slicing [92] and is therefore called *dynamic slicing*. Similar to the major objective of static slicing, dynamic slicing was specifically designed as an aid to debugging, and can be used to help in the search for offending statements which caused the program error [63].

Considerable research results on dynamic slicing of procedural programs are available [4, 52, 3, 49, 2, 78, 79, 37, 27, 98]. But dynamic slicing of object-oriented programs have scarcely been reported in the literature [100, 84, 94, 90].

Agrawal and Horgan [4] were the first to present algorithms for finding dynamic program slices using program dependence graphs. They proposed a dynamic slicing method by marking nodes on a static program dependence graph. The computed slice is not always precise, because some dependencies might not hold in dynamic execution. They also proposed a precise method based on the dynamic dependence graph (DDG) [4]. Zhao [100] extended the DDG of Agrawal and Horgan [4], known as *dynamic object-oriented dependence graph* (DODG) to represent various dynamic dependencies between statement instances for a particular execution of an object-oriented program. The DODG is an arc-classified diagraph (V, A), where V is the multi-set of flow-graph vertices, and A is the set of arcs representing dynamic control dependencies and data dependencies between vertices. Zhao's construction of DODG is based on dynamic analysis of control flow and data flow of the program, and similar to those for constructing dynamic dependence graphs for procedural programs [2]. Zhao constructed the DODG by creating a new node for each occurrence of a statement in the execution trace, and creating all the dependence edges associated with the occurrence at run-time. The execution trace of the example program in Fig. 4 on input argument *argv[1] = 3*, is given in Fig. 11. Fig. 12 shows the DODG of the example program in Fig. 4 with respect to the execution trace in Fig. 11.

Zhao [100] has considered the specific features of object-oriented programs such as method calls, inheritance, polymorphism and dynamic binding etc. in his algorithm. Zhao has regarded a call statement in an object-oriented program as one of the following statements:

– a statement that calls a free standing procedure,

– a statement that has function application,

– a statement that creates an object,

– a statement that invokes a method, or

– a statement that returns a value to its caller.

Using similar techniques proposed by Agrawal and Horgan [4], Zhao has solved the problem of representing a call statement in the DODG.

Zhao has adopted the following concepts for dynamic slicing of object-oriented programs:

– A *slicing criterion* for an object-oriented program is of the form *(s, v, t, i)*, where s is a statement in the program, v is a variable used at s, and t is an execution trace of the program with input i.

– A *dynamic slice* of an object-oriented program on a given slicing criterion *(s, v, t, i)* consists of all statements in the program that actually affected the value of a variable v at statement s.

Based on the DODG, Zhao has used a two-phase algorithm to compute dynamic slices of object-oriented programs. Computation of dynamic slices using the DODG is carried out as a graph-reachability problem. The two phases of the algorithm are:

1. Computing a dynamic slice over the DODG of the object-oriented program.
   (This can be done by using a usual depth-first or breadth-first graph traversal algorithm to traverse the DODG of the program by taking the vertex corresponding to the statement of interest as the start point of traversal.)

2. Mapping the slice over the DODG to the source code to obtain a dynamic slice of the program.
   (This can be done by simply defining a mapping function.)

It may be noted that the dynamic slice computed by Zhao [100] is not executable. This is in contrast to that presented in [52] which defines a dynamic slice as an executable subprogram. For program debugging and testing, a non-executable dynamic slice can also supply enough information as an executable dynamic slice, but can be computed more easily.

Fig. 13 shows the dynamic slice of the example program in Fig. 4 with respect to the slicing criterion *(39, current_floor, t, argv[1] = 3)*, where t is the execution trace given in Fig. 11. The statements within the boxes are included in the slice. It can be marked that the size of the resulting dynamic slice is reduced significantly compared to its corresponding static slice. The disadvantage of Zhao's approach is that the number of nodes in a DODG is equal

```
34(0)    main(argc, char  **argv)
35(0)    if   (argv[1])
37(0)    e_ptr = new Elevator(10);
2(0)     Elevator(int 1_top_floor);
3(0)     current_floor = 1;
4(0)     current_direction  = UP;
5(0)     top_floor = 1_top_floor;
38(0)    e_ptr          -.-.-.-.-.-> go(3);
15(0)       virtual void go(int floor);
16(0)    if (current _direction  = UP)
17(0)    while ( (current_floor  !=  floor) && (current_floor) <= top_floor));
18(0)     add(current_floor, 1),       ;
21(0)     add(int &a, const  int  &b);
22(0)      a = a + b;
17(1)    while ( (current_floor  !=  floor) && (current_floor) <= top_floor));
18(1)     add(current_floor, 1);
21(1)     add(int &a, const  int  &b);
22(1)      a = a + b;
17(2)    while ( (current_floor  !=  floor) && (current_floor) <= top_floor));
39(0)    cout  <<  "\n currently on floor:" << e_ptr          -.-.-.-.-.-> which_floor()  <<"
11(0)    int which_floor ( );
12(0)    return current_floor;
```

Figure 11: An execution trace of the example program in Fig. 4 on input argv[1] = 3.

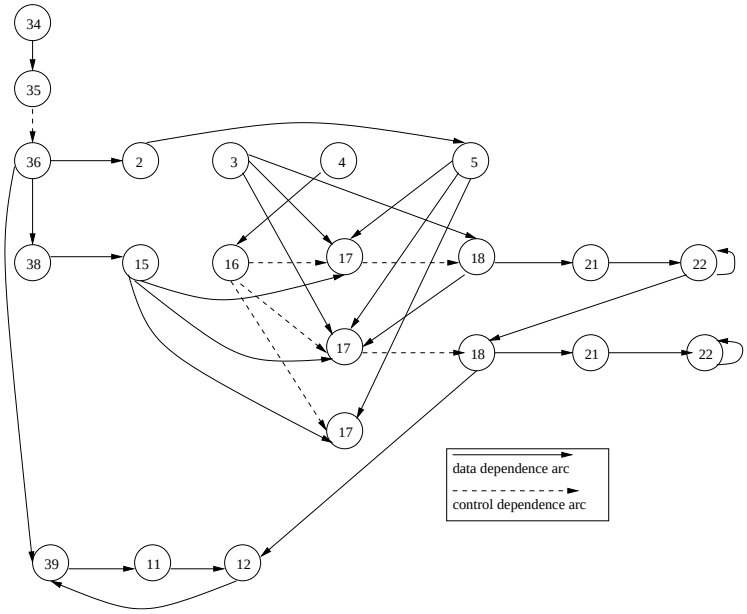

Figure 12: The DODG of the program of Fig. 4 on input argv[1]=3

to the number of executed statements, which may be unbounded for programs having many loops. Further, Zhao has used trace files to store the execution history which is expensive. The space complexity and the time complexity of this dynamic slicing algorithm are of $O(S)$ and $O(S^2)$, respectively, where S is the length of execution of the program.

Song et al.[84] proposed a method to compute forward dynamic slice of object-oriented programs using dynamic object relationship diagram (DORD). In this method, they computed the dynamic slices for each statement immediately after the statement is executed. When the last statement is executed, the dynamic slices of all executed statements have been obtained. However, only some special statements in the loops need to compute dynamic slices. So the dynamic slices computed by this technique is unnecessarily expensive.

Xu et al. [94] extended their earlier method [18] to dynamically slice object-oriented programs. Their method uses object program dependence graph (OPDG) and other static information to reduce the information to be traced during execution. Their method computes dynamic slices

```
1:  class Elevator{
        public:
2:      Elevator(int 1_top_floor)            /* initialization for Elevator */
3:        { current_floor = 1;
4:          current_direction = UP;
5:          top_floor = 1_top_floor; }                  /* end of Elevator */
6:      virtual  ~Elevator() { }
7:      void up()
8:         { current_direction = UP; }
9:      void down()
10:        {  current_direction = DOWN;  }
11:   int which_floor()
12:     { return current_floor;  }
13:     Direction direction()
14:       { return current_direction;  }
15:   virtual void go(int floor)           /* declaration for method go() */
16:     { if  (current_direction = UP )
17:         { while (current_floor != floor)
                 &&  (current_floor <= top_floor)
18:           add(current_floor, 1); }
              else
19:           { while (current_floor != floor)
                   && (current_floor  > 0 )
20:             add(current_floor, −1);  } /* end if */
          };

        private:
21:     add(int &a, const int &b)          /* This  method computes value of current_floor */
22:       {  a = a+b; } ;
          protected:
            int current_floor;
            Direction current_direction;
             int top_floor;
          };
23:     class AlarmElevator: public Elevator    /* AlarmElevator is derived from  Elevator */
          public:
24:         AlarmElevator(int top_floor);
25:           Elevator(top_floor)
26:             {alarm_on = 0;   }
27:         void set_alarm()
28:             {alarm_on = 1;  }
29:         void reset_alarm()
30:             {alarm_on = 0  }
31:         void go(int floor)
32:           { if (! alarm_on)
33:             Elevator :: go(floor)
                };
           protected:
              int alarm_on;
           };
34: main(int argc, char  **argv)  {
          Elevator *e_ptr;
35:     if (argv[1])
36:       e_ptr = new Elevator(10);
          else
37:       e_ptr = new AlarmElevator(10);
38:     e_ptr −> go(3);          /* polymorphic method call */
39:     cout << "\n currently on floor:"
              << e_ptr −> which_floor();
        }/* end of main */
```

Figure 13: The dynamic slice of the example program in Fig. 4 on slicing criterion (39, current_floor, t, argv[1] = 3).

by combining static dependence information and dynamic execution of the program. By analyzing the control flow graph of the given program, fewer breakpoints are inserted to trace the execution of the program. It is an approach combining forward analysis with backward one. In the forward process, it marks nodes on the OPDG and computes intermediate dynamic slices (which are used to record dynamic execution information) at the necessary points during the program execution. In the backward process, it traverses the OPDG marked to obtain the final dynamic slice. Based on this model, they have proposed algorithms to dynamically slice methods, objects and classes.

Wang et al. [90] presented a new dynamic slicing algorithm for Java programs which operates on compact byte code traces. According to their algorithm, first, the byte code stream corresponding to an execution of a Java program is compactly represented. Then, they perform a backward traversal of the compressed program trace to compute data/control dependencies on-the-fly. The slice is updated as these dependencies are encountered during the traversal.

The compactness of the trace representation is owing to several factors. First, byte codes which do not correspond to memory read/write (i.e., data transfer to and from the heap) or control transfer are not stored in the trace. These byte codes can be ignored for computing control and data dependencies. Secondly, the sequence of addresses used by

each memory reference, control transfer byte code is stored separately. Since these sequences typically have high repetition of pattern, they exploit such repetition to save space. They have extended the dynamic slicing algorithm to explain certain classes of omission errors.

The important advantage of their technique is that it is more space efficient than that of Zhao [100] since they use the results from data compression to compactly represent byte code traces of Java programs. The major space savings come from the optimized representation of data (instruction) addresses used by memory reference (branch) byte codes as operands. Also, their algorithm can directly traverse the compact traces without restoring to costly decompression. The disadvantage of this approach is that it uses trace files, which are expensive to handle.

Mohapatra et al. [70, 72] proposed a new algorithm for dynamic slicing of object-oriented programs. They have used *extended system dependence graph* (ESDG) as the intermediate representation. They have statically constructed the ESDG only once before the execution of the program starts. Their algorithm is based on marking and unmarking the edges of the ESDG as and when the dependencies arise and cease during run-time. So, they have named their algorithm *edge marking dynamic slicing* (EMDS) algorithm for object-oriented programs. The EMDS algorithm marks an edge of the ESDG when the corresponding dependency arises and unmarks an edge when the dependency ceases to exist. Mohapatra et al. [70, 74] also proposed another algorithm called *node marking dynamic slicing* (NMDS) algorithm for object-oriented programs. The NMDS algorithm also uses ESDG as the intermediate representation. The NMDS algorithm is based on marking and unmarking the executed nodes of the ESDG appropriately during runtime. The space complexity of both the algorithms (EMDS and NMDS) is $O(n^2)$, where n is the number of statements in the program. The time complexity of both the algorithms (EMDS and NMDS) is $O(n^2S)$, where S is the length of the execution trace. Each vertex of ESDG is annotated with its most recent dynamic slice during execution of program. Thus, slices can be extracted in constant time i.e., in $O(1)$ time.

The advantage of both the algorithms [72, 74] compared to the related ones [100, 84, 94, 90] is that they do not not require any new nodes to be created and added to the intermediate representation at run-time nor do they require to maintain any execution trace in trace files. This saves the expensive node creation and file I/O steps. Another important advantage of their algorithms is that when a request for a slice is made, it is already available. Once a slicing command is given the algorithms produce results almost instantus through a mere table-lookup and avoid on-demand slicing computation. They have shown that the EMDS and NMDS algorithms are more space and time efficient than the related algorithms [100, 84, 94, 90]. They have also shown that the NMDS algorithm is faster than the EMDS algorithm. Table 1 shows the comparison of various dynamic slicing algorithms for object-oriented programs.

Ohata et al. [81] observed that static slicing cannot compute precise slices and dynamic slicing requires too much computation time and memory space. So, they adopted an intermediate slicing method between static and dynamic slicing called *Dependence-Catch* (DC) slicing to object-oriented programs. DC slicing method uses dynamic data dependence analysis and static control dependence analysis. Dependence-Catch slicing computes more precise slices than static slicing and needs less computation time and memory space.

4 Slicing of Concurrent Object-Oriented Programs

Concurrent object-oriented programs are becoming more popular. Many of the real life object-oriented programs are concurrent which run on different machines connected to a network. It is usually accepted that understanding and debugging of concurrent object-oriented programs are much harder compared to those of sequential programs. The nondeterministic nature of concurrent programs, lack of global states, unsynchronized interactions among objects, multiple threads of control and a dynamically varying number of objects are some reasons for this difficulty [6, 9, 8]. An increasing amount of resources are being spent in debugging, testing and maintaining these products. Slicing techniques promise to come in handy at this point. However research attempts in the program slicing area have focused attention largely on sequential programs. But research reports dealing with slicing of concurrent object-oriented programs are scarce in literature [104, 101, 102, 17, 103, 105, 82, 97, 73].

4.1 Static Slicing of Concurrent Object-Oriented Programs

Static slicing of *concurrent procedural programs* has drawn the attention of many researchers [36, 35, 38, 7]. Also, static slicing of *concurrent object-oriented programs* has been addressed by some researchers [17, 103, 102, 105, 82, 97]. Excellent surveys on static slicing of concurrent object-oriented programs can be found in [20].

Zhao et al. [104] presented a dependence based representation called the *system dependence net* (SDN) which extends the previous dependence based representations [60] to represent various dependence relationships in concurrent object-oriented programs. An SDN of a concurrent object-oriented program consists of a collection of dependence graphs each representing a main procedure, a free standing procedure, or a method in a class of the program. It also consists of some additional arcs to represent direct dependencies between a call and the called procedure/method and transitive inter-procedural data dependencies. To represent interprocess communications between different methods in a class of a concurrent object-oriented program, they have introduced a new type of program dependence arc named as *external communication depen-*

Table 1: Comparison of algorithms for dynamic slicing of object-oriented programs

Approach	Category of slice	Break points inserted	Trace files used
Zhao	backward	no	yes
Song et al.	forward	no	yes
Xu et al.	backward	yes	yes
Wang et al.	backward	no	yes
Mohapatra et al.	backward	no	no

dence arc into the SDN. An SDN can be used to represent either object-oriented features or concurrency issues in a concurrent object-oriented program.

Based on the SDN, Zhao et al. [104] have used the two-phase algorithm [48] to compute static slices of concurrent object-oriented programs such as CC++. In CC++, synchronization between different threads is realized by using a single assignment variable. Threads that share access to a single assignment variable can use that variable as a synchronization element. Their system dependence net (SDN) is an extension of the SDG of Larson and Harrold [60] and therefore can be used to represent many object-oriented features in a CC++ program. To handle concurrency issues in CC++, they used an approach proposed by Cheng [22] which was originally used for representing concurrent procedural programs with a single procedure each. However, their approach, when applied to concurrent Java programs suffers from some problems due to the fact that the concurrency models of CC++ and Java are essentially different. While Java supports monitors and some low level thread synchronization primitives, CC++ uses a single assignment variable mechanism to realize thread synchronization. This difference leads to different sets of concurrency constructs in both the languages, and therefore requires different techniques to handle concurrency issues in computing slices.

Zhao [102] has also presented a dependence-based representation called the *multi-threaded dependence graph* (MDG) to represent concurrent Java programs. The MDG is composed of a collection of *thread dependence graphs* (TDG) each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. The TDG is used to represent a single thread in a concurrent Java program and is similar to the SDG [60]. The TDG of a thread is an arc-classified diagraph that consists of a number of *method dependence graphs* each representing a method, and some special kinds of dependence arcs to represent direct dependencies between a call and the called method and transitive inter-procedural data dependencies in the thread. The method dependence graph is similar to the procedure dependence graph proposed by Horwitz [48]. To represent synchronization among threads and communication among shared objects in different threads, Zhao has used two special types of dependence arcs in the MDG. He has used *synchronization dependence arcs* to represent dependence relationships between different threads due to inter-thread synchronization and *communication dependence arcs* to represent dependence relationships between different threads due to inter-thread communication.

Zhao [102] has constructed the MDG for a complete concurrent Java program by combining the TDGs for all threads in the program at synchronization and communication points by adding synchronization and communication dependence arcs between these points. Based on the MDG, Zhao [102] has presented a two-phase algorithm for computing static slices of concurrent Java programs.

Zhao et al. [105] developed another dependence-based representation called *concurrent program dependence graph* (CPDG) to represent program dependencies in a concurrent Java program. The CPDG is a diagraph which consists of a collection of dependence graphs each representing a single method in the class. Also, it includes a few additional vertices and arcs to model parameter passing between different methods in a class, and inter-thread synchronization and communication between different threads. Zhao et al. [105] used the two phase algorithm [48] to compute static slices of concurrent Java programs.

Cheng [23] introduced an intermediate representation called *program dependence net* (PDN) for parallel and distributed programs. Cheng [23] has also discussed various possible applications of PDN including slicing concurrent programs. Cheng has defined a dynamic slicing criterion of a concurrent program as a quadruplet (*s, V, H, I*), where s is a statement in the program, V is a set of variables used at s, and H is a history of an execution of the program with input I. According to Cheng, the dynamic slice *DS(s, V, H, I)* of a concurrent program on a given slicing criterion (*s, V, H, I*) consists of all statements in the program that actually affected the beginning or end of execution of s and/or affected the values of variables in V at s in the execution with I that produced H.

All these approaches [103, 102, 105, 23] slice concurrent programs by solving a node reachability problem in the graph. A shortcoming of these algorithms is that the resulting slice is not precise since they consider that dependencies between concurrently executed statements are transitive. But, in practice, the dependencies between concurrently executed statements are not transitive due to the presence of synchronization dependence and communica-

tion dependence [17].

To get a more precise slice than that of Zhao [102] and Cheng [23], Krinke [54] introduced a slicing algorithm without synchronization. Krinke has introduced a new type of dependence called *interference dependence*, among threads. In Krinke's algorithm, the interference dependence is not transitive. So, the resulting slice is more precise. However, synchronization is widely used in concurrent programs and in some environments unavoidable. Thus Krinke's algorithm can be used only in some restricted applications.

Krinke [55] has also developed another technique for context sensitive slicing of concurrent programs. In this technique, Krinke has extended the control flow graph (CFG) and program dependence graph (PDG) [48] to represent concurrent programs with interference. This technique does not require serialization or inlining of called procedures. Nanda and Ramesh [80] have extended Krinke's technique [54] to compute static slices of concurrent programs with synchronization. In their approach, they have considered *loop-carried data dependence* while computing the slice. They have proposed some optimizations to slice more efficiently. They have claimed that it could get near linear behavior for many practical concurrent programs.

Qi and Xu [82] have developed a *task synchronization reachability graph* (TSRG) for analyzing concurrent Ada programs. Based on the TSRG, they determine the synchronization dependencies in a concurrent Ada program, and construct a new type of program dependence graph, *TSRG-based program dependence graph* (RPDG). They have discussed various applications of RPDG including program understanding, debugging, testing and software maintenance etc. A limitation of this approach is that, it does not consider the communication dependencies in a concurrent program. But, communication dependencies do exist in many practical situations and is normally unavoidable in a concurrent object-oriented program. This makes Qi and Xu's approach [82] difficult to use in many practical situations.

Chen and Xu [17] have developed *concurrent control flow graphs* (CCFG) and *concurrent program dependence graphs* (CPDG) to represent concurrent Java programs. Based on the CPDG, they proposed a static slicing algorithm for concurrent Java programs [17]. In their algorithm, they have considered the fact that the inter-thread data dependencies are not transitive. So, the resulting slice is more precise than that of Zhao [102] and Cheng [23].

All the reported approaches [104, 101, 102, 23, 54, 55, 17, 82] focus on static slicing. They have not considered the *dynamic slicing* aspects.

4.2 Dynamic Slicing of Concurrent Object-Oriented Programs

Reports on dynamic slicing of concurrent object-oriented programs are scarcely available in the literature [71, 73, 76, 77].

Mohapatra et al. [71] extended the dynamic slicing algorithm of Zhao [100] to compute dynamic slices of concurrent object-oriented programs. They have used *dynamic multi-threaded dependence graph* (DMDG) as the intermediate representation. The DMDG is an arc-classified diagraph *(V, A)*, where V is the multi-set of flow graph vertices, and A is the set of arcs representing dynamic control dependencies, data dependencies, synchronization dependencies and communication dependencies between the vertices. Based on the DMDG, they have used a two-phase algorithm to compute dynamic slices of concurrent object-oriented programs. The space complexity and the time complexity of this algorithm are of $O(S)$ and $O(S^2)$, respectively, where S is the length of the execution trace. The disadvantage of this approach is that they have used a trace file to store the execution history, which is expensive.

Mohapatra et al. [70, 73, 77] have also proposed another algorithm for dynamic slicing of concurrent Java programs without using trace files. They have used *concurrent control flow graph* (CCFG) and *concurrent system dependence graph* (CSDG) as the intermediate representations. According to their approach, first the CCFG is constructed statically. Then, the CSDG is constructed by using the CCFG. A concurrent system dependence graph (CSDG) G_C of a concurrent object-oriented program *P* is a directed graph (N_C, E_C) where each node $n \in N_C$ represents a statement in P. For *x, y* $\in N_C$, $(x,y) \in E_C$ *iff* one of the following holds:

1. *y* is *control dependent* on *x*. Such an edge is called a *control dependence edge*.
2. *y* is *data dependent* on *x*. Such an edge is called a *data dependence edge*.
3. *y* is *synchronization dependent* on *x*. Such an edge is called a *synchronization dependence edge*.
4. *y* is *communication dependent* on *x*. Such an edge is called a *communication dependence edge*.

Based on the CSDG, they have proposed a marking based dynamic slicing (MBDS) algorithm for concurrent Java programs. The MBDS algorithm is based on marking and unmarking the edges of the CSDG as and when the dependencies arise and cease during run-time. MBDS algorithm permanently marks the control dependence edges as control dependencies do not change during program execution. The algorithm considers all the data dependence edges, synchronization dependence edges and communication dependence edges for marking and unmarking during run-time. During execution of the program P, MBDS algorithm marks an edge of the CSDG when its associated dependence exists, and unmarks when its associated dependence ceases to exist. After each statement *u* is executed, MBDS algorithm unmarks all incoming *marked* dependence edges excluding the control dependence edges, associated with the object *obj*, corresponding to the *previous* execution of the statement *u*. Then, the algorithm

marks the dependence edges corresponding to the *present* execution of the statement *u*.

MBDS algorithm operates in three main stages:

Stage 1: Statically constructing the intermediate program representation graph,

Stage 2: Managing the CSDG at run-time, and

Stage 3: Computing the dynamic slice.

In the first stage of MBDS algorithm, the CCFG is constructed from a static analysis of the source code. Also at this stage, using the CCFG the static CSDG is constructed. The stage 2 of the algorithm is responsible for maintaining the CSDG during run-time. The maintenance of the CSDG at run-time involves marking and unmarking the different dependencies such as data dependencies, synchronization dependencies and communication dependencies as they arise and cease. The stage 3 is responsible for computing the dynamic slices for a given slicing criterion using the uptodate CSDG. However, the third step is simply a look up as the dynamic slice computed during run-time is already available. So, when a request for a slice is made, it is immediately obtained. The space complexity of the MBDS algorithm is $O(n^2)$, where n is the number of statements in the program. The time complexity of the MBDS algorithm is $O(n^2S)$, S being the length of the execution trace. Each node of the CSDG is annotated with its most recent dynamic slice during execution of the program. Thus, the dynamic slices can be looked up in constant time i.e., in $O(1)$ time.

The important features of the MBDS algorithm are listed below.

- It computes *correct* dynamic slices with respect to any slicing criterion.
- It can handle *inter-thread synchronization* by using primitives such as *wait*() and *notify*().
- It can handle *inter-thread communication* through *shared objects*.
- No trace files are used. All information are maintained and updated dynamically for all threads and are discarded at run-time of a program on termination of a thread.
- It does not create any additional nodes during run-time. This saves the expensive node creation steps.
- When a request for a slice is made, it is already available.
- No serialization of the events of the concurrent program is required.
- As MBDS algorithm marks an edge of the CSDG only when the dependence exists, so the *transitive problem* [17] does not arise at all. So, MBDS algorithm often results in slices that are more precise.
- It can be easily extended to compute dynamic slices of distributed object-oriented programs as each component program of the whole distributed program can be considered as a single concurrent program.

Mohapatra et al. [70, 77] have developed a slicing a tool called *Dynamic Slicer for Concurrent Object-Oriented Programs* (DSCOP) to implement the MBDS algorithm. DSCOP can compute the dynamic slice of a concurrent object-oriented program with respect to any given slicing criterion. DSCOP can handle only a subset of the Java syntax. However, the tool supports inter-thread synchronization and inter-thread communication using shared memory. The lexical analyzer, parser and semantic analyzer components of DSCOP have been implemented using *ANTLR* (Another Tool for Language Recognition) [1, 67]. During semantic analysis, the input program code is appropriately instrumented so as to facilitate computation of dynamic slices and to update other associated run-time data structures after execution of each statement, as described in the MBDS algorithm. The Compile and Execute block of DSCOP compiles and links the instrumented source code using the Java compiler.

5 Slicing of Distributed Object-Oriented Programs

As software applications grow larger and become more complex, program maintenance activities such as adding new functionalities, porting to new platforms, and correcting the reported bugs consume enormous effort. This is especially true for distributed object-oriented programs. In order to cope with this scenario, programmers need effective computer-supported techniques for decomposition and dependence analysis of programs. Program slicing is one technique for such decomposition and dependence analysis.

Many real life object-oriented programs are distributed in nature and run on different machines connected to a network. The emergence of message passing standards, such as MPI, and the commercial success of high speed networks have contributed to making message passing programming common place. Message passing programming has become an attractive option for tackling the vexing issues of portability, performance, and cost effectiveness. As distributed computing gains momentum, development and maintenance tools for these distributed systems seem to gain utmost importance.

Development of real life distributed object-oriented programs presents formidable challenge to the programmer. Distributed object-oriented programs introduce several problems which do not exist in sequential programs. The non-reproducible behaviors, non-deterministic selection of communication events, lack of global states and unsynchronized interactions among threads are some of the problems which arise in case of distributed object-oriented

programs [83]. An increasing amount of effort is being spent in debugging, testing and maintaining these products. Slicing techniques promise to come in handy at this point. Through the computation of a slice for a message passing program, one can significantly reduce the amount of code that a maintenance engineer has to analyze to achieve some maintenance tasks. However, research attempts in program slicing area have focused attention largely on sequential programs. Slicing of distributed procedural programs [26, 51, 28, 22, 50, 24, 61] has also drawn the attention of many researchers. But, research reports on slicing of distributed object-oriented programs are scarcely available in the literature [34, 75].

Goel et al. [34] proposed compression schemes for representing execution profiles of shared memory parallel programs. Their representation captures control flow, data flow and synchronization in the execution of a shared memory multi-threaded program running on a multiprocessor architecture. According to their approach the control and data flow of each processor is maintained individually as whole program paths (WOP). The total order of the synchronization operations executed by all processors and the annotation of each processor's WOP with synchronization counts help to capture the inter-processor communications which are protected via synchronization primitives such as *lock*, *unlock* and *barriers*. They have illustrated the applications of compact execution traces in program debugging, program comprehension, code optimization, memory layout etc. They have used trace files to store the execution history. This leads to slow I/O operations. They have considered that the communication across different threads occurs only via synchronization primitives. Communication via shared variable accesses is not explicitly represented in their method. We have considered communications among threads through shared variables as well as message passing.

Garg et al. [33] introduced the notion of a slice of a *distributed computation*. They have defined the slice of a distributed computation with respect to a global predicate, as a computation which captures *those and only those* consistent *cuts* of the original computation which satisfy the global predicate. A *computation slice* differs from a *dynamic slice* in that it is defined for a property rather than a set of variables of a program. Unlike a program slice, which always exists, a computation slice may not always exist. They have proved that the slice of a distributed computation with respect to a predicate exists iff the set of consistent cuts that satisfy the predicate, forms a sub lattice of the lattice of consistent cuts. Mittal and Garg [68, 69] presented an efficient algorithm to *graft* two slices, that is, given two slices, either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices.

Mohapatra et al. [75] were the first to propose an algorithm for dynamic slicing of distributed object-oriented programs. They have introduced the notion of *distributed program dependence graph* (DPDG) as the intermediate program representation. In distributed object-oriented programs, communication dependency may exist among sub programs running on different machines. A *rcvmsg*() call executed on one machine, might have a pairing *sndmsg*() on some other remote machine. To represent this aspect, they have introduced a logical (dummy) node in the DPDG. They have named this logical node as a *C-node*. They have defined a C-node in the following way:

Let G_{D_1} and G_{D_2} be the DPDGs of two sub programs P_1 and P_2 respectively. Let x be a node in G_{D_1} representing a statement invoking a *sndmsg*() method. Let y be a node in G_{D_2} representing the statement invoking the corresponding *rcvmsg*() method. A *C-Node* represents a logical connection of the node y of DPDG G_{D_1} with the node x of the remote DPDG G_{D_2}. Node x represents the pairing of *sndmsg*() with a *rcvmsg*() call at node y. Node y is *Communication dependent* on node x.

The *C-nodes* maintain the logical connectivity among DPDGs representing different sub programs. A *C-node* does not represent any specific statement in the source code of a sub program. Rather, it encapsulates the triplet: $<$ *send_PID*,*send_node_number*, *dynamic_slice_at_send_node* $>$ representing the pairing of the components in a distributed program. Here, *send_PID* represents the *id* of the process sending the message, *send_node_number* represents the particular label number of the statement sending the message and *dynamic_slice_at_send_node* represents the dynamic slice at the sending node. *C-nodes* capture communication dependencies among the processes of different sub programs. It may be noted that the number of *C-nodes* in the DPDGs of a distributed C++ program, equals the number of *rcvmsg*() calls present in the program. In the DPDG, for a *rcvmsg*() node *x*, the corresponding *C-node* is represented as *C(x)*.

They have defined a distributed program dependence graph (DPDG) in the following way:

Let $P = (P_1, \ldots, P_n)$ be a distributed C++ program, and P_i be a sub program of P. P is represented using a set of DPDGs $(G_{D_1}, \ldots G_{D_n})$. The distributed program dependence graph (DPDG) G_{D_i} of the component-program P_i is a directed graph (N_{D_i}, E_{D_i}) where each node *n* (excepting the dummy nodes) represents a statement in P_i. For *x, y* $\in$ N_{D_i}, (y,x) $\in$ E_{D_i} *iff* any one of the following holds:

1. *y* is *control dependent* on *x*. Such an edge is called a *control dependence edge*.
2. *y* is *data dependent* on *x*. Such an edge is called a *data dependence edge*.
3. *y* is *fork dependent* on *x*. Such an edge is called a *fork dependence edge*.
4. *y* is *communication dependent* on *x*. Such an edge is called a *communication dependence edge*.

For all the nodes x, representing *rcvmsg*() calls, in the sub program P_i, a dummy node *C(x)* is created, and a corresponding dummy communication edge (x, C(x)) is added.

The set of DPDGs for each sub program of the distributed program is constructed statically only once before the execution of the distributed program starts. Based on the DPDG, Mohapatra et al. [75] have proposed an algorithm for dynamic slicing of distributed object-oriented programs. They have named their algorithm *parallel dynamic slicing* (PDS) algorithm as the algorithm can run parallely on several machines connected through a network. The PDS algorithm is based on marking and unmarking the edges of the DPDG as and when the dependencies arise and cease at run-time. To achieve fast response time, the PDS algorithm can run parallely on several machines connected through a network. For this purpose, we use local slicers at each remote machine. Our slicing algorithm in effect operates as the coordinated activities of local slicers running at the remote machines. Each local slicer contributes to the dynamic slice by determining its local portion of the global slice in a fully distributed fashion.

The PDS algorithm addresses the concurrency issues of object-oriented programs while computing the dynamic slices. It also handles the communication dependency arising due to objects shared among processes on same machine and due to message passing among processes on different machines. The space complexity of the PDS algorithm is $O(N^2)$, N being the total number of statements of the distributed program. The time complexity of the PDS algorithm is $O(N^2S)$, where S is the total length of execution of the distributed program.

The advantage of PDS algorithm is that it does not require any trace file to store the execution history. Another important advantage of their algorithm is that when a slicing command is given, the dynamic slice is extracted immediately by looking up the appropriate data structure, as it is already available during run-time.

Mohapatra et al. [70] also have developed another algorithm for distributed dynamic slicing of Java programs. They have named their algorithm distributed dynamic slicing (DDS) algorithm for Java programs. To achieve fast response time, DDS algorithm can run in a fully distributed manner on several machines connected through a network, rather than running it on a centralized machine. They have used local slicers at each node in a network. A local slicer is responsible for slicing the part of the program executions occurring on the local machine.

DDS algorithm uses a modified program dependence graph (PDG) [48] as the intermediate representation. This intermediate representation is called as distributed program dependence graph (DPDG). First, the DPDG is constructed statically before run-time. DDS algorithm marks and unmarks the edges of the DPDG appropriately as and when dependencies arise and cease during run-time. Such an approach is more time and space efficient and also completely does away with the necessity to maintain a trace file. This eliminates the slow file I/O operations that occur while accessing a trace file. Another advantage of DDS algorithm is that when a request for a slice for any slicing criterion is made, the required slice is already available. This appreciably reduces the response time of slicing commands.

Mohapatra et al. [70] have developed a slicing tool to implement the DDS algorithm. The tool can compute the dynamic slice of a distributed Java program with respect to a given slicing criterion. The tool handles only a subset of Java language constructs. They have named their tool *Dynamic Slicer for Distributed Java programs* (DSDJ). To construct the intermediate graphs they have used the compiler tool *ANTLR* [1, 67]. A distributed Java program is given as the input to the ANTLR program. The ANTLR program automatically generates the DPDGs for the component programs. The lexical analyzer, parser and semantic analyzer components of DSDJ are combined and the joint component is termed as program analysis component [5]. The lexical analyzer, parser and semantic analyzer components of DSDJ have been implemented using *ANTLR* [1, 67]. During semantic analysis, the Java source code is analyzed token by token to gather the various program dependencies. The tokens are first used to construct the DCFG (Distributed Control Flow Graph). Next, using the DCFG the corresponding DPDG (Distributed Program Dependence Graph) is constructed. The source program is then automatically instrumented, by adding calls to the slicer module after every statement in the source program. After the execution of each statement, the *update_slice*() method is invoked, which marks and unmarks the edges of the DPDG appropriately and updates the dynamic slice. For storing the dynamic slice of each statement they have used a two dimensional integer array. When the dynamic slice of a particular statement is requested, the *compute_slice*() method is invoked, and it provides the dynamic slice for the given slicing criterion.

6 Conclusions

We have reviewed the recent works in the area of object-oriented program slicing including static slicing of object-oriented programs, dynamic slicing of object-oriented programs, static slicing of concurrent object-oriented programs and dynamic slicing of concurrent object-oriented programs. We have presented a brief review on slicing of distributed object-oriented programs. We have also discussed some available tools for slicing of object-oriented programs. Starting with the basic sequential program constructs researchers are now trying to address various issues of slicing distributed object-oriented programs. Since modern software products are often large and consist of millions of lines of code, processing a single data structure becomes very slow and therefore development of parallel algorithms for slicing has assumed importance.

References

[1] Antlr. http://www.antlr.org/.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 60 – 73, 1991.

[3] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589 – 616, 1993.

[4] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation, SIGPLAN Notices, Analysis and Verification*, volume 25, pages 246 – 256, White Plains, NewYork, 1990.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[6] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[7] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15:3 – 43, 1983.

[8] M. Awad and J. Ziegler. A practical approach to the design of concurrency in object-oriented systems. *Software Practice and Experience*, 27:1013 – 1034, 1997.

[9] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[10] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology, Special Issue on Program Slicing*, 40(11-12):583 – 594, 1998.

[11] D. Binkley. Computing amorphous program slices using dependence graphs and a data flow model. In *Proceedings of the ACM Symposium on Applied Computing, ACM Press*, 1999.

[12] D. Binkley and K. B. Gallagher. *Program Slicing, Advances in Computers*, volume 43. Academic Press, San Diego, CA, 1996.

[13] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40:595 – 607, 1998.

[14] J. Chen, F. Wang, and Y. Chen. An object-oriented dependency graph. Technology of Object-Oriented Languages and Systems Tools, Beijing, China, 1997.

[15] J. Chen, F. Wang, and Y. Chen. Slicing object-oriented programs. In *4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC-97 / ICSC-97)*, Hong Kong, 1997.

[16] J. T. Chen, F. J. Wang, and Y. L. Chen. Slicing object-oriented programs. In *Proceedings of the APSEC'97*, pages 395 – 404, Hongkong, China, December 1997.

[17] Z. Chen and B. Xu. Slicing concurrent Java programs. *ACM SIGPLAN Notices*, 36:41 – 47, 2001.

[18] Z. Chen and B. Xu. Slicing object-oriented Java programs. *ACM SIGPLAN Notices*, 36:33 – 40, 2001.

[19] Z. Chen, B. Xu, and H. Yang. Test coverage analysis based on program slicing. In *Proceedings of IRI*, pages 559 – 565, 2003.

[20] Z. Chen, B. Xu, and J. Zhao. An overview of methods for dependence analysis of concurrent programs. *ACM SIGPLAN Notices*, 37(8):45 – 52, 2002.

[21] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang. A novel approach for measuring class cohesion based on dependence analysis. In *Proceedings of International Conference on Software Maintenance, IEEE Press*, pages 377 – 384, 2002.

[22] J. Cheng. Slicing concurrent programs - a graph theoretical approach. In *Automated and Algorithmic Debugging, AADEBUG'93, LNCS, Springer-Verlag*, pages 223 – 240, 1993.

[23] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *International Conference on Advances in Parallel and Distributed Computing*, pages 370 – 377, 1997.

[24] J. D. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13:491 – 530, 1991.

[25] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. M. Hierons, J. R. Howroyd, L. Ourabya, and M. Ward. ConSUS: a light-weight program conditioner. *Journal of Systems and Software*, 2005.

[26] S. Danicic, Mark Harman, and Yoga Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56:307 – 313, 1995.

[27] D. M. Dhamdhere, K. Gururaja, and P. G. Ganu. A compact execution history for dynamic slicing. *Information Processing Letters*, 85:145 – 152, 2003.

[28] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Fifth Workshop on Languages and Compilers for Parallel Computing, New Haven Connecticut, LNCS Springer-Verlag*, pages 329 – 337, August 1992.

[29] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 379 – 392, San Francisco, CA, USA, 1995.

[30] I. Forgacs and A. Bertolino. Feasible test path selection by principal slicing. In *Proceedings of 6th Euoropean Software Engineering Conference*, September 1997.

[31] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Consit: a fully automated conditioned program slicer. *Software Practice and Experince*, 34:15 – 46, 2004.

[32] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, SE-17(8):751 – 761, 1991.

[33] V. K. Garg and N. Mittal. On slicing a distributed computation. In *Proceedings of 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322 – 329, 2001.

[34] A. Goel, A. RoyChoudhury, and T. Mitra. Compactly representing parallel program executions. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 191 – 202, 2003.

[35] D. Goswami and R. Mall. Fast slicing of concurrent programs. In *Sixth International Conference on High Performance Computing (HiPC), LNCS Springer-Verlag*, pages 38 – 42, December 1999.

[36] D. Goswami and R. Mall. Dynamic slicing of concurrent programs. In *Seventh International Conference on High Performance Computing (HiPC), LNCS Springer-Verlag*, pages 17 – 26, December 2000.

[37] D. Goswami and R. Mall. An efficient method for computing dynamic program slices. *Information Processing Letters*, 81:111 – 117, 2002.

[38] D. Goswami, R. Mall, and P. Chatterjee. Static slicing in unix process environment. *Software Pracice and Experience*, 30:17 – 36, 2000.

[39] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 6, 1996.

[40] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of ACM SIGSOFT*, pages 29 – 40, 1995.

[41] C. Hammer and G. Snelting. An improved slicer for Java. In *Proceedings of PASTE*, pages 107 – 112, 2004.

[42] M. Harman. Conditioned slicing supports partition testing. *Journal of Software Testing, Verification and Reliability*, 12:23 – 28, 2002.

[43] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68:45 – 64, 2003.

[44] M. Harman and S. Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5, 1995.

[45] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2:85 – 92, 2001.

[46] M. Harman, L. Hu, M. Mumro, X. Zhang, D. Binkley, and S. Danicic. Syntax-directed amorphous slicing. *Automated Software Engineering*, 11:27 – 61, 2004.

[47] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Second ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 154 – 163, December 1994.

[48] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26 – 61, 1990.

[49] M. Kamkar. *Inter Procedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linkoping University, Sweden, 1993.

[50] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *International Conference on Software Maintenance, IEEE CS Press*, pages 222 – 229, October 1995.

[51] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, 2:199 – 215, 1992.

[52] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

[53] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40:647 – 659, 1998.

[54] J. Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33:35 – 42, April 1998.

[55] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of ACM SIGSOFT Software Engineering Notes*, pages 178 – 187, 2003.

[56] A. Krishnaswamy. Program slicing: An application of program dependency graphs. Technical report, Department of Computer Science, Clemson University, August 1994.

[57] D. Kung, J. Gao, P. Hisa, and Y. Toyoshima. Change impact identification in object-oriented software maintenance. In *Proceedings of International Conference on Software Maintenance*, pages 202 – 211, September 1994.

[58] D. Kung, J. Gao, P. Hisa, and Y. Toyoshima. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994.

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

[60] L. D. Larson and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, German, March 1996.

[61] Hon. F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engineering*, 11:63 – 89, 2004.

[62] D. Liang and L. Larson. Slicing objects using system dependence graphs. In *Proceedings of International Conference on Software Maintenance*, pages 358 – 367, November 1998.

[63] A. D. Lucia. Program slicing: Methods and applications. In *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142 – 149, 2001.

[64] J. R. Lyle and M. D. Weiser. Automatic program bug location by program slicing. In *Proceedings of the second International Conference on Computers and Applications, Peking, China*, pages 877 – 882, 1987.

[65] R. Mall. *Fundamentals of Software Engineering*. Prentice Hall, India, 2nd Edition, 2003.

[66] B. A. Malloy, J. D. McGregor, and A. Krishnaswamy. An extensible program representation for object oriented software. In *Proceedings of ISFST*, pages 105 – 112, 2004.

[67] A. J. S. Mills. Antlr. The University of Birmingham, 2002.

[68] N. Mittal and V. K. Garg. Computation slicing: Techniques and theory. Technical Report, TR-PDS-2001-02, The Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2001.

[69] N. Mittal and V. K. Garg. Computation slicing: Techniques and theory. In *Proceedings of Symposium on Distributed Computing*, 2001.

[70] Durga Prasad Mohapatra. *Dynamic slicing of object-oriented programs*. PhD thesis, Indian Institute of Technology, Kharagpur, India, 2005.

[71] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. Dynamic slicing of concurrent object-oriented programs. In *Proceedings of International Conference on Information Technology: Progresses and Challenges (ITPC)*, pages 283 – 290, Kathamandu, May 2003.

[72] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. An edge marking dynamic slicing technique for object-oriented programs. In *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference, IEEE CS Press*, pages 60 – 65, September 2004.

[73] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. An efficient techinque for dynamic slicing of concurrent Java programs. In *Proceedings of Acian Applied Conference on Computing (AACC-2004), Kathmandu, LNCS Springer-Verlag*, volume 3285, pages 255 – 262, October 2004.

[74] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. A node marking dynamic slicing technique for object-oriented programs. In *Proceedings of Workshop on Software Development and Architecture (SoDA)*, pages 1 – 15, Bangalore, January 2004.

[75] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. A novel approach for dynamic slicing of distributed object-oriented programs. In *Proceedings of International Conference on Distributed Computing and Internet Technology (ICDCIT), Bhubaneswar, LNCS Springer-Verlag*, volume 3347, pages 304 – 309, December 2004.

[76] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. A novel method for computing dynamic slices of concurrent C++ programs. In *Proceedings of International Conference on Advanced Computing and Communications*, pages 744 – 750, Ahmedabad, December 2004.

[77] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. Computing dynamic slices of concurrent object-oriented programs. *Information and Software Technology*, 2005.

[78] G. B. Mund, R. Mall, and S. Sarkar. An efficient dynamic program slicing technique. *Information and Software Technology*, 44:123 – 132, 2002.

[79] G. B. Mund, R. Mall, and S. Sarkar. Computation of intraprocedural dynamic program slices. *Information and Software Technology*, 45:499 – 512, April 2003.

[80] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ACM International Symposium on Software Testing and Analysis*, August 2000.

[81] F. Ohata, K. Hirose, M. Fuji, and K. Inoue. A slicing method for object-oriented programs using dynamic light weight information. In *Eighth Asia-Pacific Software Engineering Conference (APSEC-01)*, China, 2001.

[82] X. Qi and B. Xu. Dependence analysis of concurrent programs based on rechability graph and it's applications. In *Proceedings of International Conference on Computational Science*, pages 405 – 408, 2004.

[83] M. Singhal and N. G. Sivaratri. *Advanced Concepts in Operating Systems - Distributed, Database, and Multiprocessor Operating Systems*. TATA McGRAW HILL, 2002.

[84] Y. Song and D. Huynh. *Forward Dynamic Object-Oriented Program Slicing, Application Specific Systems and Software Engineering and Technology (ASSET'99)*. IEEE CS Press, 1999.

[85] C. Steindl. *Program slicing for object-oriented programming languages*. PhD thesis, Johannes Kepler University Linz, 1999.

[86] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121 – 189, 1995.

[87] F. Tip, J. D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 179 – 197, 1996.

[88] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *Proceedings of 19th International Conference on Software Engineering*, pages 433 – 443, May 1997.

[89] M. Jeffrey Voas and Gary McGraw. *Software fault-injection: inoculating programs against errors*. Wiley and Sons, 1998.

[90] T. Wang and A. RoyChoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of IEEE International Confrence on Software Engineering*, pages 512 – 521, 2004.

[91] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[92] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446 – 452, 1982.

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

[94] B. Xu and Z. Chen. Dynamic slicing object-oriented programs for debugging. In *SCAM*, pages 115 – 122, 2002.

[95] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1 – 36, 2005.

[96] L. Xu, B. Xu, Z. Chen, J. Jiang, H. Chen, and H. Yang. Regression testing for web applications based on slicing. In *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference, IEEE CS Press*, pages 652 – 656, 2003.

[97] J. Zeng, C. Soviani, and S. A. Edwards. Generating fast code from concurrent program dependence graph. In *Proceedings of ACM LCTES*, pages 175 – 181, 2004.

[98] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *International Conference on Software Engineering*, 2004.

[99] Y. Zhang, B. Xu, L. Shi, B. Li, and H. Yang. Modular monadic program slicing. In *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference, IEEE CS Press*, pages 66 – 71, September 2004.

[100] J. Zhao. Dynamic slicing of object-oriented programs. Technical report, Information Processing Society of Japan, May 1998.

[101] J. Zhao. Multithreaded dependence graphs for concurrent Java programs. In *Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, 1999.

[102] J. Zhao. Slicing concurrent Java programs. In *Proceedings of the 7th IEEE International Workshop on ProgramComprehension*, May 1999.

[103] J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. In *20th IEEE Annual International Computer Software and Applications Conference*, pages 312 – 320, August 1996.

[104] J. Zhao, J. Cheng, and K. Ushijima. A dependence-based representation for concurrent object-oriented software maintenance. In *Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 60 – 66, March 1998.

[105] J. Zhao and B. Li. Dependence based representation for concurrent Java programs and it's application to slicing. In *Proceedings of ISFST*, pages 105 – 112, 2004.